

EXPERT INSIGHT

Apps and Services with .NET 10

Build practical projects with Avalonia, Blazor, gRPC, GraphQL, and other enterprise technologies



Third Edition



Mark J. Price

<packt>

Apps and Services with .NET 10

Third Edition

Build practical projects with Avalonia, Blazor, gRPC, GraphQL, and other enterprise technologies

Mark J. Price



Apps and Services with .NET 10

Third Edition

Copyright © 2026 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Portfolio Director: Ashwin Nair

Relationship Lead: Nitin Nainani

Project Manager: Ruvika Rao

Content Engineer: Hayden Edwards

Technical Editor: Arjun Varma

Copy Editor: Hayden Edwards

Indexer: Rekha Nair

Proofreader: Hayden Edwards

Production Designer: Ajay Patule

Growth Lead: Priyadarshini Sharma

First published: November 2022

Second edition: December 2023

Third edition: February 2026

Production reference: 1050226

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-83546-220-1

www.packtpub.com

Contributors

About the author

Mark J. Price is a Microsoft Specialist: Programming in C# and Architecting Microsoft Azure Solutions, with over 20 years of experience. Since 1993, he has passed more than 80 Microsoft programming exams and specializes in preparing others to pass them. Between 2001 and 2003, Mark was employed to write official courseware for Microsoft in Redmond, USA. His team wrote the first training courses for C# while it was still an early alpha version. While with Microsoft, he taught “train-the-trainer” classes to get Microsoft Certified Trainers up-to-speed on C# and .NET. Mark has spent most of his career training a wide variety of students, from 16-year-old apprentices to 70-year-old retirees, with the majority being professional developers. Mark holds a BSc in Computer Science.

Thank you to all my readers. Your support means I get to write these books and celebrate your successes. Special thanks to the readers who give me actionable feedback via my GitHub repository and email, and interact with me and the book communities on Discord. You help make my books even better with every edition.

About the reviewer

Pavel P. Oleynik has a Ph.D. in Computer Science and is the author of more than 100 scientific papers in various languages. He organized and regularly conducted the International Theoretical and Practical Conference, Object Systems. He has been developing software since 1999 and has held the positions of team leader, technical leader, architect, and CTO in multibillion-dollar holdings. He is a recognized expert in areas such as heavy industry, coal industry, agricultural industry, banking, fintech, healthcare, retail, ERP/MES systems, SCM, ITSM, low-code/no-code systems, and foodtech.

I would like to express my deep gratitude to Mark for inviting me to act as technical editor of his books and Packt for organizing the process.

I would like to express my deep gratitude to my family. Daria, I am proud of you and your success. You are the most precious person I have in this world. I am sure that your perseverance and hard work will bear enormous fruit. Lena, thank you for respecting my boundless passion for information technology, even though it takes up all our family time.

For more than 25 years of hard work in the IT industry, I have been lucky enough to work side by side with hundreds of highly qualified professionals from all over the world. Your unlimited knowledge, which you share, motivates me to constantly explore new things. Thank you for

that and for the many hours we spent together rolling out new releases on seemingly hopeless projects. Millions of people successfully use the results of our work. The thought of this warms up in moments of IT desperation.

For my friends and parents who do not know English, I want to express my words of gratitude in the language of my countryman, the great Russian writer Anton Pavlovich Chekhov:

Для моих друзей и родителей, которые не знают английского языка, я хотел бы выразить слова благодарности на языке своего земляка, великого русского писателя Антона Павловича Чехова.

Благодарю свою маму Светлану Александровну и папу Петра Павловича, за то, что вы всегда давали мне возможность самостоятельно принимать решения и делать собственный выбор по всем ключевым вопросам в моей жизни и за то, что принимаете меня таким, какой я есть. Я понимаю, что вам достался не лучший сын на этом свете, но зато мне достались самые лучшие родители из всех возможных.

Выражаю благодарность своим верным друзьям: Вите, Диме, Саше и Тимофею. Мы так давно с вами дружим, что к нашим

*именам уже прочно приклеился префикс ‘Дядя’,
да и я не помню в своей жизни того времени,
когда вас не было рядом со мной. Я уверен, мы
будем существовать вечно.*

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow this QR code:

https://packt.link/apps_and_services_dotnet10



Preface

There are programming books that are thousands of pages long that aim to be comprehensive references to .NET-based websites, web services, and desktop and mobile apps. This book is different.

This book is a step-by-step guide to learning modern technologies for building apps and services with .NET. It is concise and aims to be a brisk, fun read that is packed with practical hands-on walkthroughs of each topic.

The breadth of the overarching narrative comes at the cost of some depth, but you will find many signposts to explore further if you wish.

In my experience, the hardest part of learning a new technology is getting started. Once I have had the most important key concepts explained and seen some practical code in action, I then feel comfortable going deeper by exploring the official documentation on my own. You can feel confident experimenting on your own once you have seen how the basics work correctly.

This book is best for those who already know the fundamentals of C# and .NET, and programmers who have worked with C# in the past but feel left behind by the changes in the past few years.

If you already have experience with older versions of the C# language and .NET libraries, then I have covered what is new in modern .NET since 2016 in an online-only section at the end of *Chapter 1, Introducing Apps and Services with .NET*.

I will call out the most important aspects of app models and frameworks for building modern user interfaces and implementing services, so you can participate in conversations with colleagues about technology and architectural choices and get productive with their implementation fast.

Who this book is for

This book is for .NET developers interested in exploring more specialized libraries and implementation fundamentals behind building services and apps.

You'll need to know your way around .NET and C# to make the most of this book, so if you want to work your way up to this book, you can pick up my other .NET book, *C# 14 and .NET 10 – Modern Cross-Platform Development Fundamentals*, first.

What this book covers

Introduction

Chapter 1, Introducing Apps and Services with .NET, is about setting up your development environment to use Visual Studio, VS Code, or Rider. You will set up SQL Server in a Docker container on Windows, macOS, or Linux. You will then execute a script to create an example database for a fictional organization named Northwind. You will create class libraries to define an EF Core model to work with the Northwind database. These class libraries are then used in most of the subsequent chapters.

You will also learn about some good places to look for help, and ways to contact me (the author of this book) to get help with an issue or give me feedback to improve the book.

Online-only sections review the new features added to the language and libraries in modern C# and .NET, and how to benchmark the performance of your code.

Apps

Chapter 2, Building Mobile Apps Using .NET MAUI, introduces you to building cross-platform mobile apps for Android and iOS using .NET MAUI. You will learn the basics of XAML, which can be used to define the user

interface for a graphical app in .NET MAUI and similar systems like Windows Presentation Foundation (WPF) and Avalonia.

Chapter 3, Building Desktop Apps Using Avalonia, introduces you to building cross-platform desktop apps for Windows, macOS, and Linux using Avalonia. I chose Avalonia because .NET MAUI does not support building Linux desktop apps, and serious real-world desktop apps like Rider are built using Avalonia, so it is a proven platform.

Chapter 4, Building Web Apps Using Blazor, is about building web user interface components for apps using Blazor.

Specialized Libraries

Chapter 5, Implementing Popular Third-Party Libraries, discusses the packages and types that allow your apps to perform common practical tasks, like:

- Formatting text and numbers using Humanizer
- Manipulating images with ImageSharp
- Logging with Serilog
- Mapping objects to other objects with AutoMapper or custom extension methods
- Making unit test assertions with FluentAssertions
- Validating data with FluentValidation
- Generating PDFs with QuestPDF

Chapter 6, Handling Dates, Times, and Internationalization, covers the types that allow your code to perform common tasks like handling dates and times, time zones, and globalizing and localizing data and the user interface of an app for internationalization. To supplement the built-in data and time types, we look at the benefits of using the much better Noda Time third-party library.

Data

Chapter 7, Managing Relational Data Using SQL, is about SQL-based relational databases like SQL Server, MySQL, and PostgreSQL, although the practical tasks use the SQL Server database that you created in *Chapter 1, Introducing Apps and Services with .NET*. You will learn how to read and write at a low-level using `SqlClient` ADO.NET libraries for maximum performance, and then use the object-to-data store mapping technology named **Dapper** for ease of development.

Chapter 8, Building Entity Models Using EF Core, is about using the higher-level object-to-data store mapping technology named **Entity Framework Core (EF Core)**. You will learn how to manage relational data stored in SQL Server, how changes to entities are tracked in the data context, and how to store entity models that use inheritance hierarchies using three different mapping strategies.

Services

Chapter 9, Building a Custom LLM-based Chat Service, covers how to build a custom chat service that integrates a Large Language Model-based (LLM) artificial intelligence with .NET projects. You will also learn how to build a Model Context Protocol (MCP) server that can be used to extend the capabilities of an LLM in a standardized way.

Chapter 10, Building and Securing Minimal API Web Services, introduces the simplest way to build web services using ASP.NET Core Minimal API. This avoids the need for controller classes. You will learn how to improve startup time and resources using native AOT publish. You then learn how to protect and secure a web service using rate limiting, CORS, and authentication and authorization. You will explore ways to test a web service using the HTTP Editor in Visual Studio and the REST Client extension for VS Code.

Chapter 11, Caching, Queuing, and Resilient Background Services, introduces service architecture design, adding features to services that improve scalability and reliability, like caching and queuing, how to handle transient

problems, and how to implement long-running services by implementing background services.

Chapter 12, Broadcasting Real-Time Communication Using SignalR, introduces you to SignalR, a technology that enables a developer to create a service that can have multiple clients and broadcast messages to all of them or a subset of them live in real-time. For example, notification systems and dashboards that need instant up-to-date information like stock prices.

Chapter 13, Combining Data Sources Using GraphQL, introduces building services that provide a simple single endpoint for exposing data from multiple sources to appear as a single combined source of data. You will use the ChilliCream GraphQL platform to implement the service, including using Hot Chocolate to build the service, Nitro to try it out, and Strawberry Shake to build a strongly-typed client. You will also learn how to implement paging, filtering, sorting, mutations, and subscriptions.

Chapter 14, Building Efficient Microservices Using gRPC, introduces building microservices using the efficient gRPC standard. You will learn about the `.proto` file format for defining services contracts and the protobuf binary format for message serialization. You will also learn how to enable web browsers to call gRPC services using gRPC JSON transcoding, how to improve the startup and memory footprint of a gRPC service with native AOT publish, handling custom data types including non-supported types like decimal, and implementing interceptors and handling faults.

Conclusion

Epilogue, describes your options for further study about building apps and services with C# and .NET, and what you should learn to become a well-rounded professional .NET developer.

Appendix A, Answers to the Test Your Knowledge Questions, has the answers to the test questions at the end of each chapter.

Appendix B, Setting Up Your Development Environment, has step-by-step

instructions for setting up your development environment. This includes a code editor such as Visual Studio or VS Code, and a database named Northwind on a database server such as SQL Server in Docker, SQL Server locally, or Azure SQL Database in the cloud.

Appendix C, Looking For Help, is all about how to find quality information about programming on the web. You will learn about Microsoft Learn documentation, including its new MCP server for integration with AI systems, getting help while coding and using dotnet commands, getting help from fellow readers in the book's Discord channel, searching the .NET source code for implementation details, and finally making the most of modern AI tools like GitHub Copilot.

The *Appendix* chapters are available to download from a link in the README file in the GitHub repository: <https://github.com/markjprice/apps-services-net10>.

You can also access a free PDF copy of the book (containing *Appendix A, B, and C*) through the following link: <https://packtpub.com/unlock>. Search for this book by name, ensure it's the correct edition, and have your invoice ready.

To get the most out of this book

You can develop and deploy C# and .NET apps and services using Visual Studio, or VS Code and the command-line tools on most operating systems, including Windows, macOS, and many varieties of Linux. An operating system that supports VS Code and an internet connection is all you need to complete this book. If you prefer to use a third-party tool like Rider, then you can.

Download the example code files

You can download or clone solutions for the step-by-step guided tasks and exercises from the GitHub repository at the following link: <https://>

github.com/markjprice/apps-services-net10.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing>. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter (X) handles. For example; “The Controllers, Models, and Views folders contain ASP.NET Core classes and the .cshtml files for execution on the server.”

A block of code is set as follows:

```
// Storing items at index positions.  
names[0] = "Kate"; names[1] = "Jack";  
names[2] = "Rebecca"; names[3] = "Tom";
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are highlighted:

```
// Storing items at index positions.  
names[0] = "Kate"; names[1] = "Jack";  
names[2] = "Rebecca"; names[3] = "Tom";
```

Any command-line input or output is written as follows:

```
dotnet new console
```

Bold: Indicates a new **term**, an important **word**, or words that you see on the screen, for example, in menus or dialog boxes. For example: “Clicking on the **Next** button moves you to the next screen.”

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book or have any general feedback, please email us at `customercare@packt.com` and mention the book’s title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit <http://www.packt.com/submit-errata>, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit <http://authors.packt.com/>.

Free benefits with your book

This book comes with free benefits to support your learning. Activate them now for instant access (see the “*How to Unlock*” section for instructions).

Here's a quick overview of what you can instantly unlock with your purchase:



DRM-Free PDF with Appendices

Download DRM-free PDF and ePub copies of this book including appendices.



7-Day Packt Library Access

Get 7-day unlimited access to 8,000+ books and videos. No credit card required.

Available for first-time Packt+ trial users only.



Next-Gen Reader Access

Read this book on Packt Reader with progress sync, dark mode and note-taking.

How to Unlock

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.



UNLOCK NOW

Note: Keep your invoice handy. Purchases made directly from Packt don't require one

Share your thoughts

Once you've read *Apps and Services with .NET 10, Third Edition*, we'd love to hear your thoughts! Please [click here to go straight to the Amazon review page](#) for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

1

Introducing Apps and Services with .NET

In this first chapter, the goals are to introduce this book, related books, and their content, set up your development environment to use Visual Studio, VS Code, or Rider, and understand your choices for building apps and services. We will also review how to use the GitHub repository and good places to look for help.

The GitHub repository for this book has solutions using full application projects for all code tasks: <https://github.com/markjprice/apps-services-net10/>.

After going to the GitHub repository, simply press the . (dot) key on your keyboard or change .com to .dev to change the repository into a live code editor based on VS Code using GitHub Codespaces.

VS Code in a web browser is great to run alongside your chosen code editor as you work through the book's coding tasks. You can compare your code to the solution code and easily copy and paste parts if needed.

Throughout this book, I use the term **modern .NET** to refer to .NET 10 and its predecessors like .NET 5, which come from .NET Core. I use the term **legacy .NET** to refer to .NET Framework, Mono, Xamarin, and .NET Standard.

Modern .NET is a unification of those legacy platforms and standards.

This chapter covers the following topics:

- Introducing this book and its contents
- App and service technologies
- Setting up your development environment
- Structuring projects and managing packages
- Making good use of the GitHub repository for this book
- Where to go for help

Free Benefits with Your Book

Your purchase includes a free PDF copy of this book – including *Appendix A, B, and C* – along with other exclusive benefits. Check the *Free Benefits with Your Book* section in the Preface to unlock them instantly and maximize your learning experience.

Introducing this book and its contents

This book caters to two audiences:

- Readers who have completed my book for beginners, *C# 14 and .NET 10 – Modern Cross-Platform Development Fundamentals*, or an earlier edition, and now want to take their learning further.
- Readers who already have basic skills and knowledge about C# and .NET and want to learn practical skills and knowledge to build real-world applications and services using popular libraries and databases.

Companion books to continue your learning journey

This is one of four books in a quartet that provides a learning journey through .NET 10:

1. The first book, *C# 14 and .NET 10 - Modern Cross-Platform Development Fundamentals*, covers the fundamentals of the C# language,

the .NET libraries, and modern ASP.NET Core for web development. It is designed to be read linearly because skills and knowledge from earlier chapters build up and are needed to understand later chapters.

2. The second book, *Real-World Web Development with .NET 10*, covers mature and proven web development technologies like ASP.NET Core MVC and controller-based Web API web services, as well as OData, FastEndpoints, and Umbraco CMS for building real-world web projects on .NET 10. You will learn how to test your web services using xUnit and test the user interfaces of your websites using Playwright, and then how to containerize your projects ready for deployment.
3. The third book, *Apps and Services with .NET 10* (the one you're reading now), covers how to build graphical user interfaces for websites, desktop, and mobile apps with Blazor, Avalonia, and .NET MAUI respectively. Then you will learn more specialized library topics like internationalization and popular third-party packages including Serilog and NodaTime. You will learn how to build native AOT-compiled services with ASP.NET Core Minimal API and how to improve performance, scalability, and reliability using caching, queues, and background services. You will implement more services using GraphQL, gRPC, and SignalR, as well as learn how to integrate LLMs to add intelligence to your solutions.
4. The fourth book, *Tools and Skills for .NET 10*, covers important tools and skills that a professional .NET developer should have. These include design patterns and solution architecture, debugging, memory analysis, all the important types of testing whether unit, performance, or web testing, and then containerization for deployment topics like Docker and Aspire. Finally, we look at how to prepare for an interview to get the .NET developer career that you want.

A summary of the .NET 10 quartet and their most important topics is shown in *Figure 1.1*:



C# language, including new C# features, debugging, unit testing, and object-oriented programming.

.NET libraries, including number types, text, regular expressions, collections, file I/O, and data with EF Core and SQLite.

Modern websites and web services with ASP.NET Core, Blazor, and Minimal API web services.

Mature websites using ASP.NET Core MVC and Umbraco CMS, including defining routes, controllers, models, and views.

Mature web services using Web API controllers, OData, and FastEndpoints, including authentication, authorization, and integration testing.

Caching, web testing, configuration, and containerizing for deployment.

Modern apps: .NET MAUI mobile apps, Avalonia desktop apps, and Blazor web apps.

Modern services: Minimal API web services, caching, queuing, GraphQL, gRPC, and SignalR.

Libraries: Third-party packages, integrating LLMs, and internationalization.

Data: SQL Server, Dapper, and EF Core.

Tools: IDEs, debugging, memory analysis, and AI assistants.

Libraries: Cryptography, multi-tasking and concurrency.

Tests: Unit, integration, performance, security, and web, including DI and IoC.

Develop: Docker and .NET Aspire.

Design: Patterns, principles, software and solution architecture.

Career: Teamwork and interviews.

Figure 1.1: Companion books for learning .NET 10

Cloud hosting and deployment topics are now online-only

The most frequent criticism I heard about the previous edition is its coverage of Azure. Although Azure has many free trials for its features, many readers would still prefer not to have to sign up for an Azure account. Azure is just one of many cloud hosts that readers will use, and more recently, some organizations have been moving back away from cloud hosting to self-hosting due to the often high costs of the cloud.

For this third edition, I decided to move all Azure cloud-specific content to online sections. They are still available for those readers who want to use Azure, but everyone else can avoid them if they want.

I made a similar choice about deployment tools. There are so many different tools that a reader might choose to use, that I decided to focus on the concept of containerization, primarily with Docker, which is often the first step to modern deployments.

What you will learn in this book

After this introductory chapter, this book can be divided into four parts, and the individual components are shown in *Figure 1.2*:

1. **App user interface technologies:** How to build user interfaces for desktop, web, and mobile apps using Avalonia, Blazor, and .NET MAUI.
2. **Specialized libraries:** Dates, times, and internationalization; and third-party libraries for image handling, data validation rules, logging, generating PDFs, and so on. These chapters can be treated like a cookbook of recipes. If you are not interested in any topic, you can skip it, and you can read them in any order.
3. **Databases:** How to store and manage data with SQL-based databases like SQL Server. Later chapters use the sample database and entity models that you will create at the end of this chapter. There is also an optional online-only chapter about Azure Cosmos DB.
4. **Service technologies:** How to build and secure services with ASP.NET Core Minimal API web services, GraphQL, gRPC, and SignalR. To improve service scalability and reliability, we cover queues, caching, and resilience features. We also cover how to integrate **Large Language Models (LLMs)** and **Model Context Protocol (MCP)** servers with your apps and services to provide AI capabilities. There is also an optional online-only chapter about Azure Functions.

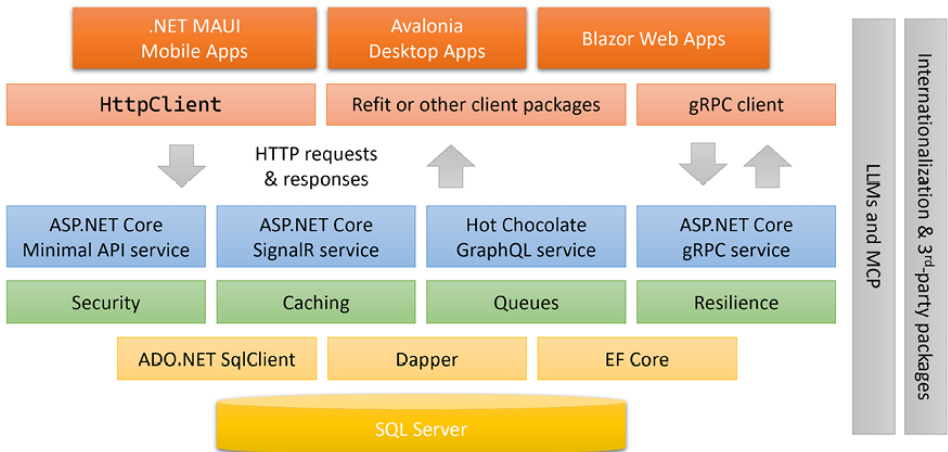


Figure 1.2: Technologies covered in Apps and Services with .NET 10

My learning philosophy

Most people learn complex topics best by imitation and repetition rather than reading a detailed explanation of the theory; therefore, I will not overload you with detailed explanations of every step throughout this book. The idea is to get you to write some code and see it run.

You don't need to know all the nitty-gritty details immediately. That will be something that comes with time as you build your own apps and go beyond what any book can teach you.

If you have an issue with something in this book, then please contact me before resorting to a negative review on Amazon. Authors cannot respond to Amazon reviews, so I cannot contact you to resolve the problem. I want to help you get the best from my book, and I want to listen to your feedback and do better in the next edition. Please email me (my email address can be found in the GitHub repository for the book), chat with me in

the Discord channel for the book at the following link https://packt.link/apps_and_services_dotnet10, or raise an issue in the book's GitHub repository at the following link: <https://github.com/markjprice/apps-services-net10/issues>.

App and service technologies

In this section, we'll look at the major app and service technologies in the .NET ecosystem, from building traditional Windows-only desktop apps to creating cross-platform experiences. We'll explore frameworks like ASP.NET Core for web and API development, introduce service technologies such as WCF and newer alternatives like gRPC and GraphQL, and discuss options for building apps that go beyond Windows, including .NET MAUI and competing cross-platform frameworks. The goal is to understand the landscape of tools available so you can make informed decisions about which approach best fits your project.

Microsoft calls platforms for building applications and services **app models** or **workloads**.

Understanding .NET

.NET, .NET Core, .NET Framework, and Xamarin are related and overlapping platforms for developers used to build applications and services. If you are not familiar with the history of .NET, then I will point you to each of these .NET concepts at the following link, which is from the *C# 14 and .NET 10 – Modern Cross-Platform Development Fundamentals* book: <https://github.com/markjprice/cs14net10/blob/main/docs/ch01-dotnet-history.md>.

Building websites and apps using

ASP.NET Core

Websites are made up of multiple web pages loaded statically from the filesystem or generated dynamically by a server-side technology such as ASP.NET Core. A web browser makes `GET` requests using **Uniform Resource Locators (URLs)**, which identify each page and can manipulate data stored on the server using `POST`, `PUT`, `PATCH`, and `DELETE` requests.

With many websites, the web browser is treated as a presentation layer, with almost all the processing performed on the server side. Some JavaScript might be used on the client side to implement some presentation features, such as carousels, or to perform data validation.

ASP.NET Core provides multiple technologies for building websites:

- **ASP.NET Core Razor Pages** can dynamically generate HTML for simple websites.
- **ASP.NET Core MVC** is an implementation of the **Model-View-Controller (MVC)** design pattern, which is popular for developing complex websites.
- **Razor class libraries** provide a way to package reusable functionality for ASP.NET Core projects including user interface components.
- **Blazor** lets you build user interface components using C# and .NET and then run them in a web browser or embedded web component instead of a JavaScript-based UI framework like Angular, React, and Vue.

Building web and other services

There are no formal definitions, but services are sometimes described based on their complexity:

- **Service:** All functionality needed by a client app in one monolithic service.
- **Microservice:** Multiple services that each focus on a smaller set of functionalities. The guiding principle for what the boundary of

functionality should be for a microservice is that each microservice should own its own data. Only that microservice should read/write that data. If you have a data store that multiple services access, then they are not microservices.

- **Nanoservice:** A single function provided as a service. Unlike services and microservices that are hosted 24/7/365, nanoservices are often inactive until called upon to reduce resources and costs. For this reason, they are also known as serverless services.

Although the theory of microservices and serverless services has made them a fashionable choice over the past decade or so, monolithic services have recently had a resurgence in popularity as developers have found the reality of microservices does not always match the theory.

You will learn how to build ASP.NET Core Web API and Minimal API web services that use HTTP as the underlying communication technology and follow the design principles of Roy Fielding's REST architecture. You will also learn how to build services using web and other technologies that extend basic web APIs, including:

- **gRPC:** For building highly efficient and performant microservices with support for almost any platform.
- **SignalR:** For implementing real-time communications between components.
- **GraphQL:** For letting the client control what data is retrieved across multiple data sources. Although GraphQL can use HTTP, it does not have to, and it does not follow the web design principles defined by Roy Field in his dissertation about REST APIs.
- **Azure Functions:** For hosting serverless nanoservices in the cloud. This is an online-only section.
- **odata:** For easily wrapping EF Core and other data models as a web service. This is covered in the book, *Real-World Web Development with .NET 10*.

Good practice: One of the most important service architecture principles is to make method calls chunky instead of chatty. In other words, try to bundle all the data needed for an operation in a single call, rather than requiring multiple calls to transmit all that information. This is because the overhead of a remote call is one of the biggest negative effects of services. This is also why having smaller and smaller services can have a hugely negative impact on a solution architecture.

Windows Communication Foundation

In 2006, Microsoft released .NET Framework 3.0 with some major new frameworks, one of which was **Windows Communication Foundation (WCF)**. It abstracted the business logic implementation of a service from the communication technology infrastructure so that you could easily switch to an alternative in the future or even have multiple mechanisms to communicate with the service.

WCF heavily uses XML configuration to declaratively define endpoints, including their address, binding, and contract. This is known as the ABCs of WCF endpoints. Once you understand how to do this, WCF is a powerful yet flexible technology.

Microsoft decided not to officially port WCF to modern .NET, but there is a community-owned OSS project named **CoreWCF** managed by the .NET Foundation. If you need to migrate an existing service from .NET Framework to modern .NET or build a client to a WCF service, then you can use CoreWCF. Be aware that it can never be a full port since parts of WCF are Windows-specific.

Technologies like WCF allow for the building of distributed applications. A client application can make **remote procedure calls (RPCs)** to a server

application. Instead of using a port of WCF to do this, we should use an alternative RPC technology like gRPC, which is covered in this book.

You can learn more about CoreWCF in its GitHub repository found at the following link: <https://github.com/CoreWCF/CoreWCF>. You can read the announcement about client support for calling WCF or CoreWCF with System.ServiceModel 6.0 at the following link: <https://devblogs.microsoft.com/dotnet/wcf-client-60-has-been-released/>.

Summary of choices for services

Each service technology has its pros and cons based on its feature support, as shown in *Table 1.1*:

Feature	gRPC	HTTP/2	SOAP	REST	GraphQL	JSON-RPC	AMQP	WebRTC	WebSockets	WebGL
Minimum HTTP version	1.0	2.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Browser support	Yes	Yes	No	Yes	Yes	Yes	No	Yes	Yes	Yes
WebRTC (WebRTCish)	Yes	Yes	No	Yes	Yes	Yes	No	Yes	Yes	Yes
Service API documentation	Yes	Yes	No	Yes	Yes	Yes	No	Yes	Yes	Yes
Client library	Yes	Yes	No	Yes	Yes	Yes	No	Yes	Yes	Yes
Harding	Yes	Yes	No	Yes	Yes	Yes	No	Yes	Yes	Yes

Table 1.1: Pros and cons of common service technologies

Use these recommendations for various scenarios as guidance, as shown in *Table 1.2*:

Recommendation	
Public services are best for services that need to be publicly	

	accessible, especially if they need to be called from a browser or mobile device.	
OData	Data GraphQL are both good choices for exposing complex hierarchical datasets that could come from different data stores. OData is designed and supported by Microsoft via official .NET packages. GraphQL is designed by Facebook and supported by third-party packages	
SRPC	is designed for low-latency and high-throughput communication. gRPC is great for lightweight internal microservices where efficiency is critical	
PRPC	has excellent support for bidirectional streaming. gRPC services can push messages in real time without polling. SignalR is designed for real-time communication of many kinds, so it tends to be easier to implement than gRPC, although it is less efficient	
SignalR	has great support for broadcasting real-time communication to many clients	
gRPC	to date supports all popular development languages, making gRPC a good choice for multi-language and platform environments	
gRPC	messages are constrained with Protobuf, a lightweight message format. A gRPC message is always smaller than an equivalent JSON message	
Serverless	functions do not need to be hosted 24/7 so they are a good choice for nanoservices that usually do not need to be running constantly. Amazon Web Services (AWS) Lambdas are an alternative.	

Table 1.2: Service scenarios and the recommended implementation technology

Building Windows-only apps

Technologies for building Windows-only apps, primarily for desktop, include:

- **Windows Forms**, 2002
- **Windows Presentation Foundation (WPF)**, 2006
- **Windows Store**, 2012

- **Universal Windows Platform (UWP)**, 2015
- **Windows App SDK** (formerly **WinUI 3** and **Project Reunion**), 2021

Understanding legacy Windows application platforms

With the Microsoft Windows 1.0 release in 1985, the only way to create Windows applications was to use the C language and call functions in three core DLLs named `KERNEL`, `USER`, and `GDI`. Once Windows became 32-bit with Windows 95, the DLLs were suffixed with 32 and became known as **Win32 API**.

In 1991, Microsoft introduced Visual Basic, which provided developers with a visual, drag-and-drop-from-a-toolbox-of-controls way to build the user interface for Windows applications. It was immensely popular, and the Visual Basic runtime is still distributed as part of Windows 11 today.

With the first version of C# and .NET Framework released in 2002, Microsoft provided technology for building Windows desktop applications named **Windows Forms**. The equivalent at the time for web development was named **Web Forms**, hence the complementary names. The code could be written in either Visual Basic or C# languages. Windows Forms had a similar drag-and-drop visual designer, although it generated C# or Visual Basic code to define the user interface, which can be difficult for humans to understand and edit directly.

In 2006, Microsoft released a more powerful technology for building Windows desktop applications, named **Windows Presentation Foundation (WPF)**, as a key component of .NET Framework 3.0 alongside **WCF** and Windows **Workflow (WF)**.

Although a WPF app can be created by writing only C# statements, it can also use **eXtensible Application Markup Language (XAML)** to specify its user interface, which is easy for both humans and code to understand. Visual Studio is partially built with WPF.

In 2012, Microsoft released Windows 8 with its Windows Store apps that run in a protected sandbox.

In 2015, Microsoft released Windows 10 with an updated Windows Store app concept named **Universal Windows Platform (UWP)**. UWP apps can be built using C++ and DirectX UI, JavaScript and HTML, or C# using a custom fork of modern .NET that is not cross-platform but provides full access to the underlying WinRT APIs.

UWP apps can only execute on the Windows 10 or Windows 11 platforms, not earlier versions of Windows, but UWP apps can run on Xbox and Windows Mixed Reality headsets with motion controllers.

Many Windows developers rejected Windows Store and UWP apps because they have limited access to the underlying system. Microsoft recently created **Project Reunion** and **WinUI 3**, which work together to allow Windows developers to bring some of the benefits of modern Windows development to their existing WPF apps and allow them to have the same benefits and system integrations that UWP apps have. This initiative is now known as **Windows App SDK**.

This book does not cover Windows App SDK because it is not cross-platform. If you would like to learn more, you can start at the following link:
<https://learn.microsoft.com/en-us/windows/apps/windows-app-sdk/>.

Understanding modern .NET support for legacy Windows platforms

The on-disk size of the .NET SDKs for Linux and macOS is about 330 MB. The on-disk size of the .NET SDK for Windows is about 440 MB. This is because it includes **.NET Desktop Runtime**, which allows the legacy Windows application platforms Windows Forms and WPF to run on modern .NET.

There are many enterprise applications built using Windows Forms and WPF that need to be maintained or enhanced with new features, but until recently they were stuck on .NET Framework, which is now a legacy platform. With modern .NET and its .NET Desktop Runtime, these apps can now use the full modern capabilities of .NET. Windows desktop app developers can also optionally install the Windows Compatibility Pack. You can learn more about this at the following link: <https://learn.microsoft.com/en-us/dotnet/core/porting/windows-compat-pack>.

Building cross-platform mobile and desktop apps

There are two major mobile platforms, Apple's iOS and Google's Android, each with its own programming languages and platform APIs. There are also two major desktop platforms, Apple's macOS and Microsoft's Windows, each with its own programming languages and platform APIs, as shown in the following list:

- **iOS:** Objective-C or Swift and UIKit
- **Android:** Java or Kotlin and Android API
- **macOS:** Objective-C or Swift and AppKit or Catalyst
- **Windows:** C, C++, or many other languages and Win32 API or Windows App SDK

.NET MAUI

Cross-platform mobile and desktop apps can be built once for the **.NET Multi-platform App UI (MAUI)** platform and then can run on many mobile and desktop platforms.

.NET MAUI makes it easy to develop those apps by sharing user interface components as well as business logic; they can target the same .NET APIs used by console apps, websites, and web services.

The apps can exist standalone, but they usually call services to provide an experience that spans all your computing devices, from servers and laptops to phones and gaming systems.

.NET MAUI supports existing MVVM and XAML patterns. The team also plans to add support in the future for **Model-View-Update (MVU)** with C#, which is like Apple's SwiftUI. MVU using Comet is still only a proof of concept. It is not as mature or well supported as SwiftUI. I will not cover it in this book. You can read more about it at the following link: <https://github.com/dotnet/Comet>.

Before Microsoft created .NET MAUI, third parties created open-source initiatives to enable .NET developers to build cross-platform apps using XAML, named **Uno** and **Avalonia**.

Uno platform

Uno is “*the first C# & XAML, free and open-source platform for creating true single-source, multi-platform applications,*” as stated on its own website at the following link: <https://platform.uno/>.

Developers can reuse 99% of the business logic and UI layer across native mobile, web, and desktop platforms.

The Uno platform uses the Xamarin native platform but not Xamarin.Forms. For WebAssembly, Uno uses the mono-wasm runtime just like Blazor WebAssembly. For Linux, Uno uses Skia to draw the user interface on the canvas.

A book to read to learn about the Uno platform can be found at the following link: <https://www.packtpub.com/product/creating-cross-platform-c-applications-with-uno-platform/9781801078498>.

Avalonia

Avalonia “*is a cross-platform UI framework for .NET. It creates pixel-perfect, native apps*” and “*is supported on all major platforms.*” Avalonia “*is the trusted UI framework for complex apps,*” as stated on its official website home page at the following link: <https://avaloniaui.net/>.

You can think of Avalonia as a spiritual successor to WPF. WPF, Silverlight, and UWP developers can continue to benefit from their years of pre-existing knowledge and skills.

It was used by JetBrains to modernize their legacy WPF-based tools and take them cross-platform. This means their C# code editor runs on Windows, macOS, and Linux.

The Avalonia extension for Visual Studio and deep integration with Rider makes development easier and more productive.

Which platform is best for building web, mobile, and desktop apps?

.NET MAUI is best for building cross-platform mobile apps because it supports the two major platforms: Apple iOS and Google Android. You will be introduced to this technology in [Chapter 2, Building Mobile Apps Using .NET MAUI](#). .NET MAUI is less good for desktop apps because it only supports Windows and Mac Catalyst, but not proper Mac or Linux desktop apps.

Avalonia is best for building cross-platform desktop apps because it supports more OSes than .NET MAUI including Linux. You will be introduced to this technology in [Chapter 3, Building Desktop Apps Using Avalonia](#).

Blazor is best for building cross-platform web apps because it uses C# and .NET and supports both server-side components and WebAssembly for standalone web apps. You will be introduced to this technology in [Chapter 4, Building Web Apps Using Blazor](#).

Managing data with relational databases

Three of the most popular relational databases are:

- **Microsoft SQL Server:** A proprietary RDBMS developed by Microsoft, widely used in enterprise environments and applications running on Windows and Azure.
- **MySQL:** An open-source RDBMS owned by Oracle, commonly used in web applications, startups, and cloud-based environments.
- **PostgreSQL:** A powerful, open-source RDBMS known for its advanced features, extensibility, and compliance with SQL standards.

Typical users and use cases are shown in *Table 1.3*:

Enterprise users				
Enterprise applications, government, business intelligence (BI), insurance ERPs, Microsoft ecosystem applications				
Medium to large businesses, Word Press, developers, SaaS, CRM, high weight cloud services				
Data analysis, GIS, scientific computing, operations, financial applications				

Table 1.3: Typical users and use cases for an RDBMS

Database system considerations

An important consideration is cost:

- SQL Server requires licensing per core, which can make it expensive for high-performance deployments. SQL Server Express is free but limited. Licensing can be expensive, especially for enterprise editions.
- MySQL can be free, but Oracle’s commercial options (for example, MySQL Enterprise) add costs. It is open source with a GPL license, but Oracle offers commercial support. Enterprise edition costs can add up.
- PostgreSQL is free to use and modify, with no vendor lock-in. It is completely open-source (BSD license) with optional paid support from third-party vendors.

Another consideration is performance and scalability:

- SQL Server is highly optimized for transactional workloads and integrates well with Microsoft’s ecosystem. It is high-performance and optimized for large datasets. It scales well in enterprise settings, but licensing costs may be a barrier.
- MySQL is fast for simple queries but can struggle with complex transactions. It is especially fast for simple read-heavy workloads. It is good for horizontal scaling, but complex queries can be slow.
- PostgreSQL is designed for complex queries and supports parallelism, making it efficient at scale. It handles complex queries well, supports parallel execution, and it scales both vertically and horizontally, supports advanced partitioning.

Database system pros and cons

All three support ACID compliance for transactions, support JSON, stored procedures and triggers. In general:

- SQL Server has strong enterprise features but requires licensing for high availability.
- MySQL is simple and widely supported but lacks some advanced transactional features.
- PostgreSQL is the most feature-rich, supporting advanced data types and concurrency.

More details of pros and cons for each RDBMS are shown in *Table 1.4*:

RDBMS			
SQL Server supports for especially intelligence (BI) and analytics. Primarily optimized for Windows and Linux support was added in 2019 . Resource-heavy features like Transparent Data Encryption. Comprehensive management tools using SQL Server Management Studio and code editor integration			
MySQL supports for non-transactional data support.			

	Easy to use and easy to integrate with other applications like web applications.				
	Large community and extensive documentation.				
	Avoidable license cost and quality of stewardship of MySQL.				
	Highly SQL portable, for example, it supports custom data types and procedural languages can be complex for beginners.				
	Strong GUI management tools and a polished and polished SQL Server.				
	Open-source with no licensing restrictions.				
	Multi-version concurrency control (MVCC) based architecture improves performance in high-concurrency environments.				

Table 1.4: Pros and cons for three RDBMS

All three are supported by AWS and Azure cloud hosts. MySQL and PostgreSQL are also supported on Google Cloud.

To summarize:

- SQL Server is best for a Microsoft ecosystem and if you need enterprise-grade security and BI features.
- MySQL is best for web apps and lightweight simple applications.
- PostgreSQL is best for advanced SQL compliance and extensibility and if you want a completely free, feature-rich database.

You will learn about working with SQL databases in [Chapter 7, Managing Relational Data Using SQL](#) and [Chapter 8, Building Entity Models Using EF Core](#). There is also an online-only chapter about *Managing NoSQL Data Using Azure Cosmos DB*.

Now that we’ve reviewed the theory about the apps and services technologies that can be used with .NET 10, let’s get practical and see how you can set up your development environment.

Setting up your development environment

Before you start programming, you'll need a code editor for C#. Microsoft has a family of code editors and **Integrated Development Environments (IDEs)**, which include:

- Visual Studio for Windows
- VS Code for Windows, Mac, or Linux
- VS Code for the Web or GitHub Codespaces

Third parties have created their own C# code editors, for example, Rider, which is available for Windows, Mac, or Linux, but does have a license cost. Rider is popular with more experienced .NET developers.

Choosing the appropriate tool and application type for learning

What is the best tool and application type for building apps and services with C# and .NET?

I want you to be free to choose any C# code editor or IDE to complete the coding tasks in this book, including VS Code, Visual Studio, or even Rider.

In this book, I give general instructions that work with all tools, so you can use whichever tool you prefer.

Using Visual Studio for general development

Visual Studio can create most types of applications, including console apps, websites, web services, desktop, and mobile apps.

Although you can use Visual Studio with a .NET MAUI project to write a cross-platform mobile app, you still need macOS and Xcode to compile it.

Visual Studio only runs on Windows 10 version 1909 or later, or Windows Server 2016 or later, and only on 64-bit versions. Version 17.4 is the first version to support native Arm64.

Warning! Visual Studio for Mac does not officially

support .NET 8 or later, and it reached end-of-life in August 2024. If you have been using Visual Studio for Mac, then you should switch to VS Code for Mac, Rider for Mac, or use Visual Studio for Windows in a virtual machine on your local computer or in the cloud using a technology like Microsoft Dev Box. The retirement announcement can be read here: <https://devblogs.microsoft.com/visualstudio/visual-studio-for-mac-retirement-announcement/>.

Using VS Code for cross-platform development

The most modern and lightweight code editor to choose from, and the only one from Microsoft that is cross-platform, is VS Code. It can run on all common operating systems, including Windows, macOS, and many varieties of Linux, including **Red Hat Enterprise Linux (RHEL)** and Ubuntu.

VS Code is a good choice for modern cross-platform development because it has an extensive and growing set of extensions to support many languages beyond C#.

Being cross-platform and lightweight, it can be installed on all platforms that your apps will be deployed to for quick bug fixes and so on. Choosing VS Code means a developer can use a cross-platform code editor to develop cross-platform apps.

VS Code has strong support for web development, although it currently has weak support for mobile and desktop development.

VS Code is supported on ARM processors, so you can develop on Apple Silicon computers and Raspberry Pi.

VS Code is by far the most popular integrated development environment, with over 75% of professional developers selecting it in the Stack Overflow 2025 survey, which you can read at the following link: <https://survey.stackoverflow.co/2025/technology/#1-dev-id-es>.

What I used

To write and test the code for this book, I used the following hardware and software:

- Windows 11 on an HP Intel x64 laptop with Visual Studio, VS Code, and Rider.
- macOS on an Apple Silicon Mac mini (M1) desktop with VS Code and Rider.

I hope that you have access to a variety of hardware and software, too because seeing the differences on various platforms deepens your understanding of development challenges, although any one of the above combinations is enough to learn how to build practical apps and websites.

Getting Started: *Chapter 1* of the *C# 14 and .NET 10 – Modern Cross-Platform Development Fundamentals* book has online sections showing how to get started with multiple projects using various code editors like Visual Studio, VS Code, or Rider. You can read the sections at the following link: <https://github.com/markjprice/cs14net10/blob/main/docs/code-editors/README.md>.

Rider and its warnings about boxing

If you use Rider and you have installed the Unity Support plugin, then it will complain a lot about boxing. A common scenario when boxing happens is when

value types like `int` and `DateTime` are passed as positional arguments to string formats. This is a problem for Unity projects because they use a different memory garbage collector than the normal .NET runtime. For non-Unity projects, like all the projects in this book, you can ignore these boxing warnings because they are not relevant. You can read more about this Unity-specific issue at the following link: <https://docs.unity3d.com/Manual/performance-garbage-collection-best-practices.html#boxing>.

Deploying cross-platform

Your choice of code editor and operating system for development does not limit where your code gets deployed.

.NET 10 supports the following platforms for deployment:

- **Windows:** Windows 10 version 1607 or later, Windows 11 version 22000 or later, Windows Server 2012 R2 SP1 or later, and Nano Server version 2019 or later.
- **Linux:** Alpine Linux 3.22, Azure Linux 3.0, CentOS Stream 9 or 10, Debian 12 or 13, Fedora 42, openSUSE Leap 15.6, RHEL 9 or 10, SUSE Enterprise Linux 15.6, and Ubuntu 22.04, 24.04, or 25.04.
- **Android:** API 21 or later is the minimum SDK target. Versions 12, 12.1, 13, 14, and 15.
- **iOS and iPadOS:** 18. iOS 12.2 is used as the minimum SDK target.
- **Mac:** macOS 14 or 15.

Windows ARM64 support in .NET 5 and later means you can develop on, and deploy to, Windows ARM devices like Microsoft's Surface Pro 11 and Surface Laptop 7 and similar devices.

You can review the latest supported operating systems and versions at the following link:

<https://github.com/dotnet/core/>

[blob/main/release-notes/10.0/supported-os.md](#).

Downloading and installing Visual Studio

Many professional Microsoft developers use Visual Studio in their day-to-day development work. Even if you choose to use VS Code to complete the coding tasks in this book, you might want to familiarize yourself with Visual Studio too.

If you do not have a Windows computer, then you can skip this section and continue to the next section where you will download and install VS Code on macOS or Linux.

Since October 2014, Microsoft has made a professional quality edition of Visual Studio available to students, open-source contributors, and individuals for free. It is called Community Edition. Any of the editions are suitable for this book. If you have not already installed it, let's do so now:

1. Download the latest version of Visual Studio from the following link:
<https://visualstudio.microsoft.com/downloads/>.

At the time of publishing in February 2026, the latest version of Visual Studio is version 18.2 and it is branded as Visual Studio 2026. If you choose to do so, you should be able to use Visual Studio 2022 version 17.14 or later to complete this book, although the user interface might move things around a bit.

1. Start the installer.
2. On the **Workloads** tab, select the following:
 - **ASP.NET and web development**
 - **Azure development**

- **.NET Multi-platform App UI development**
 - **.NET desktop development** (because this includes console apps)
 - **Desktop development with C++** with all default components (because this enables publishing console apps and web services that start faster and have smaller memory footprints)
8. On the **Individual components** tab, in the **Code tools** section, select **Git for Windows**.
 9. Click **Install** and wait for the installer to acquire the selected software and install it.
 10. When the installation is complete, click **Launch**.
 11. The first time that you run Visual Studio, you will be prompted to sign in. If you have a Microsoft account, you can use that account. If you don't, then register for a new one at the following link: <https://signup.live.com/>.
 12. The first time that you run Visual Studio, you will be prompted to configure your environment. For **Development Settings**, choose **Visual C#**. For the color theme, I chose **Light**, but you can choose whatever tickles your fancy.
 13. If you want to customize your keyboard shortcuts, navigate to **Tools | Options...**, and then select the **Environment | Keyboard** option.

Visual Studio keyboard shortcuts

In this book, I will avoid showing keyboard shortcuts since they are often customized. Where they are consistent across code editors and commonly used, I will try to show them.

If you want to identify and customize your keyboard shortcuts, then you can, as shown at the following link: <https://learn.microsoft.com/en-us/visualstudio/ide/identifying-and-customizing-keyboard-shortcuts-in-visual-studio>.

Downloading and installing VS Code

VS Code has rapidly improved over the past couple of years and has pleasantly surprised Microsoft with its popularity. If you are brave and like to live on the bleeding edge, then there is the **Insiders** edition, which is a daily build of the next version.

Even if you plan to only use Visual Studio for development, I recommend that you download and install VS Code and try the coding tasks in this chapter using it, and then decide if you want to stick with just using Visual Studio for the rest of the book.

Let's now download and install VS Code, the .NET SDK, and the C# Dev Kit extension:

1. Download and install either the Stable build or the Insiders edition of VS Code from the following link: <https://code.visualstudio.com/>.

If you need more help installing VS Code on any operating system, you can read the official setup guide at the following link: <https://code.visualstudio.com/docs/setup/setup-overview>.

1. Download and install the .NET SDK for version 10.0 from the following link: <https://www.microsoft.com/net/download>.
2. To install the **C# Dev Kit** extension using the user interface, you must first launch the VS Code application.
3. In VS Code, click the **Extensions** icon or navigate to **View | Extensions**.
4. **C# Dev Kit** is one of the most popular extensions available, so you should see it at the top of the list, or you can enter `C# Dev Kit` in the search box.

C# Dev Kit has a dependency on the **C#** extension version 2.0 or later, so you do not have to install the **C#** extension separately. Note that **C#** extension version 2.0 or later no longer uses OmniSharp since it has a new **Language Service Protocol (LSP)** host. **C# Dev Kit** also has dependencies on the **.NET Install Tool for Extension Authors** and **IntelliCode for C# Dev Kit** extensions so they will be installed too.

1. Click **Install** and wait for the supporting packages to download and install.

Good practice: Be sure to read the license agreement for **C# Dev Kit**. It has a more restrictive license than the **C#** extension: <https://aka.ms/vs/csdevkit/license>.

Installing other extensions

In later chapters of this book, you will use more VS Code extensions. If you want to install them now, all the extensions that we will use are shown in *Table 1.5*:

Description	name and identifier	
C# Dev Kit extension from Microsoft. Manage your code with a solution explorer and test your code with integrated unit test discovery and execution.		
Includes the C# extension		
C# editing support, including syntax highlighting, IntelliSense, Go To Definition, Find All References, debugging support for .NET, and support		

		for csproj projects on Windows, macOS, and Linux	
	GitHub Copilot Chat	AI peer programming tool that helps you write code faster and smarter	chat
	MSBuild project tools	for MSBuild project files, including autocomplete for PackageReference elements	
	SQL Server (SSQL) for VS Code	SQL Database, and SQL data warehouse everywhere with a rich set of functionalities	
	REST Client	request and view the response directly in VS Code.	
		human rest-client	
	Deploy to Azure	MSIL assemblies – support for modern .NET, .NET Framework, .NET Core, and .NET Standard	
	Python VS Code	language support for VS Code: navigation, IntelliSense, diagnostics, formatting, compilation, linting, breaking-change checks, and schema graphs.	

Table 1.5: VS Code extensions used in this book

Managing VS Code extensions at the command prompt

You can install a VS Code extension at the command prompt or terminal, as shown in Table 1.6:

Description		
View installed extensions		
Install the specified extension	<extension-id>	
Uninstall the specified extension	<extension-id>	

Table 1.6: Working with extensions at the command prompt

For example, to install the **C# Dev Kit** extension, enter the following at the command prompt:

```
code --install-extension ms-
```

I have created PowerShell scripts to install and uninstall the VS Code extensions in the preceding table. You can find them at the following link:
<https://github.com/markjprice/apps-services-net10/tree/main/scripts/extension-scripts>.

Understanding VS Code versions

Microsoft releases a new feature version of VS Code (almost) every month and bug-fix versions more frequently. For example:

- Version 1.107.0, November 2025 feature release
- Version 1.107.1, November 2025 bug fix release

The version used in this book is 1.108.2, the December 2025 bug fix release, but the version of Microsoft VS Code is less important than the version of the **C# Dev Kit** or **C#** extension that you install. I recommend **C# Dev Kit** v1.90.2 or later with **C#** extension v2.112.45 or later.

While the C# extension is not required, it provides IntelliSense as you type, code navigation, and debugging features, so it's something that's very handy to install and keep updated to support the latest C# language features.

VS Code keyboard shortcuts

In this book, I will avoid showing keyboard shortcuts used for tasks like creating a new file since they are often different on different operating systems. The situations where I will show keyboard shortcuts are when you need to repeatedly press the key, for example, while debugging. These are also more likely to be consistent across operating systems.

If you want to customize your keyboard shortcuts for VS Code, you can do so,

as shown at the following link: <https://code.visualstudio.com/docs/getstarted/keybindings>.

I recommend that you download a PDF of keyboard shortcuts for your operating system from the following list:

- Windows: <https://code.visualstudio.com/shortcuts/keyboard-shortcuts-windows.pdf>
- macOS: <https://code.visualstudio.com/shortcuts/keyboard-shortcuts-macos.pdf>
- Linux: <https://code.visualstudio.com/shortcuts/keyboard-shortcuts-linux.pdf>

Using other project templates

When you install the .NET SDK, there are many project templates included. Let's review them:

1. At a command prompt or terminal, enter the following command:

```
dotnet new list
```

.NET 7 and later SDKs support either `dotnet new --list` or `dotnet new list`. The .NET 6 and earlier SDKs only support `dotnet new --list`.

1. You will see a list of currently installed templates, including templates for Windows desktop development if you are running on Windows, the most common of which are shown in *Table 1.7*:

Template Name				
NET MAUI App				
NET MAUI Blazor App				

ASP.NET Core Empty				
ASP.NET Core gRPC Service				
ASP.NET Core Web API				
ASP.NET Core Web API (native AOT)				
ASP.NET Core Web App (Model-View-Controller)				
Blazor Web App				
Class Library				
Console App				
EditorConfig File				
global.json File				
Solution File				
xUnit Test Project				

Table 1.7: Project template full and short names

.NET MAUI projects are not supported for Linux. The team has said they have left that work to the open-source community. If you need to create a truly cross-platform graphical app, then take a look at Avalonia at the following link: <https://avaloniaui.net/>.

Structuring projects and managing packages

How should you structure your projects? In this book, we will build multiple projects using different technologies that work together to provide a single solution. With large, complex solutions, you will often have many projects that need the same packages, and updating their versions every time a new version is released is a pain. Let’s review how we can manage this.

Project naming and port numbering

conventions

If you complete all the coding tasks in this book, then you will end up with dozens of projects. Many of those will be websites and services that require port numbers for hosting on the `localhost` domain.

With large, complex solutions, it can be difficult to navigate the entire code. So, a good reason to structure your projects well is to make it easier to find components. It is good to have an overall name for your solution that reflects the application or solution.

In the 1990s, Microsoft registered **Northwind** as a fictional company name for use in database and code samples. It was first used as the sample database for their Access product and then also used in SQL Server. We will build multiple projects for this fictional company, so we will use the name `Northwind` as a prefix for all the project names.

Good practice: There are many ways to structure and name projects and solutions, for example, using a folder hierarchy as well as a naming convention. If you work in a team, make sure you know how your team does it.

It is good to have a naming convention for your projects in a solution so that any developer can tell what each one does instantly. A common choice is to use the type of project, for example, class library, console app, website, and so on, as shown in *Table 1.8*:

Name		Description
A class library project for common types like interfaces, enums, classes, records, and structs, used across multiple projects		
A class library project for code that uses EF Core entity models. Entity models are often used on both the server and client side, so it is best to separate dependencies on specific database providers		

	Access library project for the EF Core database context with dependencies on specific database providers		
	An ASP.NET Core project for an HTTP API service. A good choice for integrating with websites because it can use any JavaScript library or Blazor to interact with the service		
	A client web application. The last part of the name indicates that it is a console app		
	An ASP.NET Core project for a gRPC service.		

Table 1.8: Example naming conventions for common project types

To enable you to run any of these projects simultaneously, we must make sure that we do not configure duplicated port numbers. I have used the following convention:

```
https://localhost:5[chapternumber]1/
http://localhost:5[chapternumber]2/
```

For example, for the encrypted connection to the website built in [Chapter 10, Building and Securing Minimal API Web Services](#), I used port 5101, as shown in the following link:

```
https://localhost:5101/.
```

The folder structure that we will use in this book is shown in *Figure 1.3*:

apps-services-net10

AzureExtras

HttpRequests

Internationalization

ModernApps

ModernServices

PopularPackages

Directory.Packages.props

nuget.config

Ch1-4, 7-9

ChatApp
Northwind.Console.EFCore
Northwind.DataContext
Northwind.EntityModels.tests

Ch5

FluentTests
FluentValidation.Validators
GeneratingPdf.Models
MappingObjects.Mappers
Serilogging

Ch6

TestingWithTimeProvider
WorkingWithNodaTime
Internationalization.slnx

Ch10 to 14

Northwind.Background.Hangfire
Northwind.Common
Northwind.GraphQL.Service
Northwind.MinimalApi.Service
Northwind.SignalR.Client.Console
Northwind.WebApi.Client.Console
Northwind.WebApi.Service

LlamaApp
Northwind.Console.HierarchyMapping
Northwind.DesktopApp
Northwind.Mau.Client

FluentValidation.Console
GeneratingPdf.Console
HumanizingData
MappingObjects.Models
WorkingWithImages

TimeFunctionsLib
WorkingWithTime

Northwind.Background.Workers
Northwind.GraphQL.Client.Console
Northwind.Grpc.Client.Mvc
Northwind.Queue.Consumer
Northwind.SignalR.Client.Console.Streams
Northwind.WebApi.Client.Mvc
ModernServices.slnx

Northwind.Blazor
Northwind.Console.SqlClient
Northwind.EntityModels
ModernApps.slnx

FluentValidation.Models
GeneratingPdf.Document
MappingObjects.Console
MappingObjects.Tests
PopularPackages.slnx

WorkingWithCultures
WorkingWithTimeZones

Northwind.Client.Resilient
Northwind.GraphQL.Client.Mvc
Northwind.Grpc.Service
Northwind.Queue.Models
Northwind.SignalR.Service.Client.Mvc
Northwind.WebApi.Controllers

Figure 1.3: Folder structure for all the solutions and projects created in this book

Central Package Management

By default, with the .NET SDK CLI and most code editor-created projects, if you need to reference a NuGet package, you add the reference to the package name and version directly in the project file, as shown in the following markup:

```
<ItemGroup> <PackageReference  
  Include="Microsoft.EntityFrameworkCore.SqlServer"  
  Version="10.0.0" /> ... </ItemGroup>
```

Central Package Management (CPM) is a feature that simplifies the management of NuGet package versions across multiple projects and solutions within a directory hierarchy. This is particularly useful for large solutions with many projects, where managing package versions individually can become cumbersome and error-prone.

Features and benefits of CPM

The key features and benefits of CPM include:

- **Centralized Control:** CPM allows you to define package versions in a single file, typically `Directory.Packages.props`, which is placed in the root directory of a directory hierarchy that contains all your solutions and projects. This file centralizes the version information for all NuGet packages used across the projects in your solutions.
- **Consistency:** Ensures consistent package versions across multiple projects. Having a single source of truth for package versions eliminates discrepancies that can occur when different projects specify different versions of the same package.
- **Simplified Updates:** Updating a package version in a large solution becomes straightforward. You update the version in the central file, and all projects referencing that package automatically use the updated version. This significantly reduces the maintenance overhead.
- **Reduced Redundancy:** Removes the need to specify package versions in individual project files (`.csproj`). This makes project files cleaner and easier to manage, as they no longer contain repetitive version information.

Good practice: It is important to regularly update NuGet packages and their dependencies to address security vulnerabilities.

Defining project properties to reuse version numbers

Microsoft packages usually have the same number each month, like 10.0.2 in February, 10.0.3 in March, and so on. You can define properties at the top of your `Directory.Packages.props` file and then reference these properties throughout the file. This approach keeps package versions consistent and makes updates easy.

For example, at the top of the `Directory.Packages.props` file, within a `<ProjectGroup>` tag, define your custom property and then reference it for

the package version, as shown in the following markup:

```
<Project> <PropertyGroup>
<MicrosoftPackageVersion>10.0.2</
MicrosoftPackageVersion> </PropertyGroup>
<ItemGroup> <PackageVersion
Include="Microsoft.EntityFrameworkCore"
Version="$(MicrosoftPackageVersion)" />
<PackageVersion
Include="Microsoft.Extensions.Logging"
Version="$(MicrosoftPackageVersion)" /> <!--
Add more Microsoft packages as needed. --> </
ItemGroup> <!-- Other packages with specific
versions. --> <ItemGroup> <PackageVersion
Include="Newtonsoft.Json" Version="13.0.4" />
</ItemGroup> </Project>
```

Note the following about the preceding configuration:

- **Define a property:** In the `<PropertyGroup>` element, the line `<MicrosoftPackageVersion>10.0.2</MicrosoftPackageVersion>` defines the property. This value can be changed once at the top of the file, and all references will update automatically.
- **Reference the property:** Use the syntax `$(PropertyName)` to reference the defined property. All occurrences of `$(MicrosoftPackageVersion)` will resolve to the version number that you set.

When the monthly update rolls around, for example, from 10.0.2 to 10.0.3, you only have to update this number once.

You might want separate properties for related packages if they differ in version number, such as:

```
<AspNetCorePackageVersion>10.0.3</  
AspNetCorePackageVersion>  
<EFCorePackageVersion>10.1.2</  
EFCorePackageVersion>
```

This allows independent updates if packages later diverge in their release cycles or versions.

After making changes, at the terminal or command prompt, run the following command:

```
dotnet restore
```

This will verify the correctness of your references and quickly alert you if you've introduced errors. By adopting this pattern combined with CPM, you simplify version management, reduce redundancy, and make your projects easier to maintain over time.

Good practice: Choose clear and consistent property names, like `MicrosoftPackageVersion` or `AspNetCorePackageVersion`, to easily distinguish between different package ecosystems. Check your `Directory.Packages.props` file into source control. Regularly update and test after changing versions to ensure compatibility.

Configuring CPM for this book's projects

Let's set up Central Package Management for a solution that we will use throughout the rest of the chapters in this book. Let's go:

1. Create a new folder named `apps-services-net10` that we will use

for all the code in this book. For example, on Windows, create a folder:
`C:\apps-services-net10`.

Good practice: For all the projects that you create for this book, keep your root path short and avoid using `#` in your folder and file names, or you might see compiler errors like `RSG002: TargetPath not specified for additional file`. For example, do *not* use `C:\My C# projects\` as your root path!

1. In the `apps-services-net10` folder, create a new file named `Directory.Packages.props`. At the command prompt or terminal, you can use the following command: `dotnet new packagesprops`

To save you time manually typing this large file, you can download it at the following link: <https://github.com/markjprice/apps-services-net10/blob/main/code/Directory.Packages.props>.

1. In `Directory.Packages.props`, modify its contents, as shown in the following markup:

```
<Project> <PropertyGroup>
<ManagePackageVersionsCentrally>true </
ManagePackageVersionsCentrally>
<Net10>10.0.2</Net10> <Avalonia>11.3.11</
Avalonia> <AI>10.2.0-preview.1.26063.2</
```

```
AI> <Hangfire>1.8.22</Hangfire>
<HotChocolate>15.1.11</HotChocolate> </
PropertyGroup> <ItemGroup Label="Common
packages in multiple chapters."> <!--For
binding configuration from
appsettings.json and environment
variables--> <PackageVersion
Include="Microsoft.Extensions.Configuration.Binder
Version="$(Net10)" /> <PackageVersion
Include="Microsoft.Extensions.Configuration.Json"
Version="$(Net10)" /> <PackageVersion
Include=
"Microsoft.Extensions.Configuration.EnvironmentVa
Version="$(Net10)" /> <!--For HTTP hosting
and resilience.--> <PackageVersion
Include="Microsoft.Extensions.Hosting"
Version="$(Net10)" /> <PackageVersion
Include="Microsoft.Extensions.DependencyInjection
Version="$(Net10)" /> <PackageVersion
Include="Microsoft.Extensions.Http"
Version="$(Net10)" /> <PackageVersion
Include="Microsoft.Extensions.Http.Resilience"
Version="10.2.0" /> <!--For logging to
console.--> <PackageVersion
Include="Microsoft.Extensions.Logging.Console"
Version="$(Net10)" /> <PackageVersion
Include="Microsoft.Extensions.Logging.Debug"
Version="$(Net10)" /> <PackageVersion
Include="Spectre.Console" Version="0.54.0"
/> <!--For unit testing.-->
<PackageVersion
Include="coverlet.collector"
Version="6.0.4" /> <PackageVersion
Include="Microsoft.NET.Test.Sdk"
Version="18.0.1" /> <PackageVersion
```

```
Include="xunit" Version="2.9.3" />
<PackageVersion
Include="xunit.runner.visualstudio"
Version="3.1.5" /> <!--For JSON
processing.--> <PackageVersion
Include="Newtonsoft.Json" Version="13.0.4"
/> </ItemGroup> <ItemGroup Label= "Chapter
1 Introducing Apps and Services with
.NET"> <PackageVersion
Include="BenchmarkDotNet" Version="0.15.8"
/> </ItemGroup> <ItemGroup Label= "Chapter
2 Building Mobile Apps Using .NET MAUI.">
<PackageVersion
Include="Microsoft.Maui.Controls"
Version="10.0.20" /> <PackageVersion
Include="CommunityToolkit.Mvvm"
Version="8.4.0" /> <PackageVersion
Include="CommunityToolkit.Maui"
Version="13.0.0" /> </ItemGroup>
<ItemGroup Label= "Chapter 3 Building
Desktop Apps Using Avalonia.">
<PackageVersion Include="Avalonia"
Version="$(Avalonia)" /> <PackageVersion
Include="Avalonia.Desktop"
Version="$(Avalonia)" /> <PackageVersion
Include="Avalonia.Themes.Fluent"
Version="$(Avalonia)" /> <PackageVersion
Include="Avalonia.Fonts.Inter"
Version="$(Avalonia)" /> <PackageVersion
Include="Avalonia.Diagnostics"
Version="$(Avalonia)" /> </ItemGroup>
<ItemGroup Label="Chapter 4 Building Web
Apps Using Blazor."> <PackageVersion
Include=
"Microsoft.AspNetCore.Components.WebAssembly.Serv
```



```

Version="$(Net10)" /> <PackageVersion
Include=
"Microsoft.AspNetCore.Components.WebAssembly"
Version="$(Net10)" /> <PackageVersion
Include=
"Microsoft.AspNetCore.Components.WebAssembly.DevS
Version="$(Net10)" /> <PackageVersion
Include="Microsoft.AspNetCore.Components.QuickGrid
Version="$(Net10)" /> <PackageVersion
Include="Radzen.Blazor" Version="8.6.0" />
</ItemGroup> <ItemGroup Label= "Chapter 5
Using Popular Third-Party Libraries">
<PackageVersion
Include="SixLabors.ImageSharp"
Version="3.1.12" /> <PackageVersion
Include="Humanizer" Version="3.0.1" />
<PackageVersion Include="Serilog"
Version="4.3.0" /> <PackageVersion
Include="Serilog.Sinks.Console"
Version="6.1.1" /> <PackageVersion
Include="Serilog.Sinks.File"
Version="7.0.0" /> <PackageVersion
Include="Serilog.Sinks.Async"
Version="2.1.0" /> <!-- AutoMapper 15 or
later requires a licence. -->
<PackageVersion Include="AutoMapper"
Version="14.0.0" /> <PackageVersion
Include="Mapster" Version="7.4.0" /> <!--
FluentAssertions 8.0.0 and later have
restricted the licence.--> <!--Minimum
7.2.0 (inclusive) up to maximum 8.0.0
(exclusive).--> <PackageVersion
Include="FluentAssertions"
Version="[7.2.0,8.0.0)" /> <PackageVersion
Include="FluentValidation"

```

```
Version="12.1.1" /> <!-- The newest
version with an MIT license (07/02/2024).
--> <PackageVersion Include="QuestPDF"
Version="2022.12.15" /> <!-- A 2023.* or
later version requires setting the
Settings.License property. --> <!--
<PackageReference Include="QuestPDF"
Version="2025.7.1" /> --> </ItemGroup>
<ItemGroup Label= "Chapter 6 Handling
Dates, Times, and Internationalization">
<!-- The newest version before the
controversy. --> <PackageVersion
Include="Moq" Version="4.18.4" />
<PackageVersion
Include="Microsoft.Extensions.Localization"
Version="$(Net10)" /> <PackageVersion
Include="NodaTime" Version="3.3.0" /> </
ItemGroup> <ItemGroup Label= "Chapter 7
Managing Relational Data Using SQL">
<PackageVersion
Include="Microsoft.Data.SqlClient"
Version="6.1.3" /> <PackageVersion
Include="Npgsql" Version="10.0.1" />
<PackageVersion Include="Dapper"
Version="2.1.66" /> </ItemGroup>
<ItemGroup Label= "Chapter 8 Building
Entity Models Using EF Core">
<PackageVersion
Include="Microsoft.EntityFrameworkCore.Design"
Version="$(Net10)" /> <PackageVersion
Include="Microsoft.EntityFrameworkCore.SqlServer"
Version="$(Net10)" /> <PackageVersion
Include="Npgsql.EntityFrameworkCore.PostgreSQL"
Version="10.0.0" /> </ItemGroup>
<ItemGroup Label= "Chapter 9 Building an
```

```

LLM-Based Chat Service"> <PackageVersion
Include="Microsoft.Extensions.AI.OpenAI"
Version="$(AI)" /> <PackageVersion
Include="Microsoft.Extensions.AI"
Version="10.1.1" /> <PackageVersion
Include="Microsoft.Extensions.DataIngestion"
Version="$(AI)" /> <PackageVersion
Include="Microsoft.Extensions.DataIngestion.Markdown"
Version="$(AI)" /> <PackageVersion
Include="PdfPig" Version="0.1.13" />
<PackageVersion
Include="Microsoft.ML.Tokenizers.Data.Cl100kBase"
Version="1.0.1" /> <PackageVersion
Include="Microsoft.ML.Tokenizers.Data.O200kBase"
Version="1.0.1" /> <PackageVersion
Include=
"Microsoft.SemanticKernel.Connectors.SqliteVec"
Version="1.67.1-preview" />
<PackageVersion
Include="ModelContextProtocol"
Version="0.5.0-preview.1" />
<PackageVersion Include="OllamaSharp"
Version="5.4.12" /> </ItemGroup>
<ItemGroup Label= "Chapter 10 Building and
Securing Minimal API Web Services">
<PackageVersion
Include="Microsoft.AspNetCore.OpenApi"
Version="$(Net10)" /> <PackageVersion
Include="Scalar.AspNetCore"
Version="2.12.6" /> <PackageVersion
Include="AspNetCoreRateLimit"
Version="5.0.0" /> <PackageVersion
Include="Microsoft.AspNetCore.Authentication.JwtBearer"
Version="$(Net10)" /> </ItemGroup>
<ItemGroup Label="Chapter 11 Caching,

```

```
Queuing, and Resilient Background
Services"> <PackageVersion
Include="Microsoft.Extensions.Caching.Hybrid"
Version="10.2.0" /> <PackageVersion
Include="Microsoft.Extensions.Http.Polly"
Version="$(Net10)" /> <PackageVersion
Include="Polly.Contrib.WaitAndRetry"
Version="1.1.1" /> <PackageVersion
Include="RabbitMQ.Client" Version="7.2.0"
/> <PackageVersion Include="Hangfire.Core"
Version="$(Hangfire)" /> <PackageVersion
Include="Hangfire.SqlServer"
Version="$(Hangfire)" /> <PackageVersion
Include="Hangfire.AspNetCore"
Version="$(Hangfire)" /> </ItemGroup>
<ItemGroup Label="Chapter 12 Broadcasting
Real-Time Communication Using SignalR">
<PackageVersion
Include="Microsoft.AspNetCore.SignalR.Client"
Version="$(Net10)" /> </ItemGroup>
<ItemGroup Label="Chapter 13 Combining
Data Sources Using GraphQL">
<PackageVersion
Include="HotChocolate.AspNetCore"
Version="$(HotChocolate)" />
<PackageVersion
Include="HotChocolate.Data.EntityFramework"
Version="$(HotChocolate)" />
<PackageVersion
Include="StrawberryShake.Server"
Version="$(HotChocolate)" /> </ItemGroup>
<ItemGroup Label="Chapter 14 Building
Efficient Microservices Using gRPC">
<PackageVersion Include="Google.Protobuf"
Version="3.33.4" /> <PackageVersion
```

```

Include="Grpc.AspNetCore" Version="2.76.0"
/> <PackageVersion
Include="Grpc.Net.ClientFactory"
Version="2.76.0" /> <PackageVersion
Include="Grpc.Tools" Version="2.76.0" />
<PackageVersion
Include="Microsoft.AspNetCore.Grpc.JsonTranscoding"
Version="$(Net10)" /> </ItemGroup>
<ItemGroup Label="Chapter X1 Managing
NoSQL Data Using Azure Cosmos DB">
<PackageVersion
Include="Microsoft.Azure.Cosmos"
Version="3.53.1" /> <PackageVersion
Include="Gremlin.net" Version="3.7.4" />
</ItemGroup> <ItemGroup Label="Chapter X2
Building Serverless Nanoservices Using
Azure Functions"> <PackageVersion
Include="NCrontab.Signed" Version="3.4.0"
/> <PackageVersion
Include="Microsoft.Azure.Functions.Extensions"
Version="1.1.0" /> <PackageVersion
Include="Microsoft.Azure.Functions.Worker"
Version="2.1.0" /> <PackageVersion
Include="Microsoft.Azure.Functions.Worker.Sdk"
Version="2.0.5" /> <PackageVersion
Include=
"Microsoft.Azure.Functions.Worker.Extensions.Http"
Version="3.3.0" /> <PackageVersion
Include=
"Microsoft.Azure.Functions.Worker.Extensions.Timer"
Version="4.3.1" /> <PackageVersion
Include=
"Microsoft.Azure.Functions.Worker.Extensions.Storage"
Version="5.5.3" /> <PackageVersion
Include=

```

```
"Microsoft.Azure.Functions.Worker.Extensions.Storage"
Version="6.8.0" /> <PackageVersion
Include="SixLabors.ImageSharp.Drawing"
Version="2.1.7" /> </ItemGroup> </Project>
```

Warning! The

<ManagePackageVersionsCentrally> element and its `true` value must go all on one line. Also, you cannot use floating wildcard version numbers like `10.0-*` as you can in an individual project. Wildcards are useful to automatically get the latest patch version, for example, monthly package updates on Patch Tuesday. But with CPM you must manually update the versions.

For any projects that we add underneath the folder containing this file, we can reference the packages without explicitly specifying the version, as shown in the following markup:

```
<ItemGroup> <PackageReference
Include="Microsoft.EntityFrameworkCore.SqlServer"
/> <PackageReference
Include="Microsoft.EntityFrameworkCore.Design"
/> </ItemGroup>
```

CPM good practice

You should regularly review and update the package versions in the `Directory.Packages.props` file to ensure that you are using the latest stable releases with important bug fixes and performance improvements.

Good practice: I recommend that you set a monthly event in your calendar for the second Wednesday of each month. This will occur after the second Tuesday of each month, which is Patch Tuesday when Microsoft releases bug fixes and patches for .NET and related packages.

For example, throughout 2026 there are likely to be new versions each month, so you can go to the NuGet page for each of your packages. You can then update individual package versions if necessary, for example, as shown in the following markup:

```
<ItemGroup Label="For EF Core.">
  <PackageVersion
    Include="Microsoft.EntityFrameworkCore.SqlServer"
    Version="10.0.3" />
```

Or, if you have defined custom properties referenced by multiple packages, update those version numbers, as shown highlighted in the following markup:

```
<PropertyGroup>
  <ManagePackageVersionsCentrally>true </
  ManagePackageVersionsCentrally>
  <Net10>10.0.3</Net10>
```

Before updating package versions, check for any breaking changes in the release notes of the packages. Test your solution thoroughly after updating to ensure compatibility.

Educate your team and document the purpose and usage of the `Directory.Packages.props` file to ensure everyone understands how to manage package versions centrally.

In each individual project file you can override an individual package version by using the `VersionOverride` attribute on a `<PackageReference />` element, as shown highlighted in the following markup:

```
<PackageReference
  Include="Microsoft.EntityFrameworkCore.SqlServer"
  VersionOverride="10.0.0" />
```

This can be useful if a newer version introduces a regression bug so you can force the use of an older version without the bug until the bug is fixed in a later patch version.

Warning! If you see error NU1008 The following `PackageReference` items cannot define a value for `Version`: `PackageName`. Projects using Central Package Management must define a `Version` value on a `PackageVersion` item., then remove the version attribute from the package reference in your project file. You can learn more at the following link: <https://learn.microsoft.com/en-us/nuget/reference/errors-and-warnings/nul008>.

You can learn more about CPM at the following link: <https://learn.microsoft.com/en-us/nuget/consume-packages/central-package-management>.

Package Source Mapping

If you use CPM and you have more than one package source configured for your code editor, as shown in *Figure 1.4*, then you will see NuGet Warning NU1507. For example, if you have both the default package source (<https://api.nuget.org/v3/index.json>) and a custom package source configured:

There are 2 package sources defined in your configuration. When using central package management, please map your package sources with package source mapping (<https://aka.ms/nuget-package-source-mapping>) or specify a single package source. The following sources are defined: <https://api.nuget.org/v3/index.json>, <https://contoso.myget.org/F/development/>.

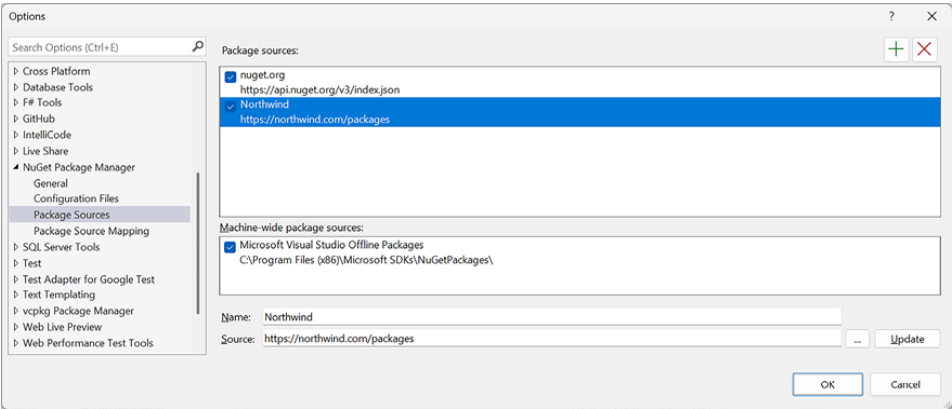


Figure 1.4: Visual Studio with two NuGet package sources configured

NU1507 warning reference page can be found at the following link: <https://learn.microsoft.com/en-us/nuget/reference/errors-and-warnings/nu1507>.

Understanding Package Source Mapping

Package Source Mapping (PSM) can help safeguard your software supply chain if you use a mix of public and private package sources as in the preceding example.

By default, NuGet will search all configured package sources when it needs to download a package. When a package exists on multiple sources, it may not be deterministic which source the package will be downloaded from. With PSM, you can filter, per package, which source(s) NuGet will search.

PSM is supported by Visual Studio 2022, .NET 6 and later, and NuGet 6 and later. Older tooling will ignore the PSM configuration.

Now let's see how to enable PSM.

Enabling Package Source Mapping

To enable PSM, you must have a `nuget.config` file.

Good practice: Create a `nuget.config` file at the root of your source code directory hierarchy.

In a PSM file, there are two parts: defining package sources, and mapping package sources to packages. All requested packages must map to one or more sources by matching a defined package pattern. In other words, once you have defined a `packageSourceMapping` element, you must explicitly define which sources every package – including transitive packages – will be restored from.

For example, if you want most packages to be sourced from the default `nuget.org` site, but there are some private packages that must be sourced from your organization's website, you would define the two package sources, and set the mapping (assuming all your private packages are named `Northwind.Something`), as shown in the following markup:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration> <packageSources> <!-- <clear /
> ensures no additional sources are inherited
from another config file. --> <clear /> <!--
key can be any identifier for your source. -->
<add key="nuget.org" value="https://
api.nuget.org/v3/index.json" /> <add
key="Northwind" value="https://northwind.com/
packages" /> </packageSources> <!-- All
packages sourced from nuget.org except
Northwind packages. --> <packageSourceMapping>
<!-- key value for <packageSource> should
match key values from <packageSources> element
--> <packageSource key="nuget.org"> <package
pattern="*" /> </packageSource> <packageSource
key="Northwind"> <package
pattern="Northwind.*" /> </packageSource> </
packageSourceMapping> </configuration>
```

Let's create a `nuget.config` for all the solutions and projects in this book that will use `nuget.org` as the source for all packages:

1. In the `apps-services-net10` folder, create a new file named `nuget.config`.
2. In `nuget.config`, modify its contents, as shown in the following markup:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration> <!-- <clear /> ensures no
additional sources are inherited from
another config file. --> <packageSources>
<clear /> <!-- key can be any identifier
for your source. --> <add key="nuget.org"
value="https://api.nuget.org/v3/
```

```
index.json" /> </packageSources> <!-- All
packages sourced from nuget.org. -->
<packageSourceMapping> <!-- key value for
<packageSource> should match key values
from <packageSources> element -->
<packageSource key="nuget.org"> <package
pattern="*" /> </packageSource> </
packageSourceMapping> </configuration>
```

3. Save changes.

You can learn more about
nuget.config at the following link:
<https://learn.microsoft.com/en-us/nuget/reference/nuget-config-file>.

You can learn more about PSM at the
following link: <https://learn.microsoft.com/en-us/nuget/consume-packages/package-source-mapping>.

Treating warnings as errors

By default, compiler warnings may appear if there are potential problems with your code when you first build a project, but they do not prevent compilation, and they are hidden if you rebuild. Warnings are given for a reason, so ignoring warnings encourages poor development practices.

Some developers would prefer to be forced to fix warnings, so .NET provides a project setting to do this, as shown highlighted in the following markup:

```

<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup> <OutputType>Exe</OutputType>
  <TargetFramework>net10.0</TargetFramework>
  <ImplicitUsings>enable</ImplicitUsings>
  <Nullable>enable</Nullable>
  <TreatWarningsAsErrors>true</
  TreatWarningsAsErrors> </PropertyGroup>

```

I have enabled the option to treat warnings as errors in (almost) all the solutions in the GitHub repository.

The exceptions are the gRPC projects. This is due to a combination of factors. In .NET 7 or later, the compiler will warn if you compile source files that contain only lowercase letters in the name of a type. For example, if you define a person class, as shown in the following code:

```
public class person { }
```

This compiler warning has been introduced so that a future version of C# can safely add a new keyword knowing it will not conflict with the name of a type that you have used, because only C# keywords should contain only lowercase letters.

Unfortunately, the Google tools for generating C# source files from .proto files generate aliases for class names that only contain lowercase letters, as shown in the following code:

```

#region Designer generated code using pb =
global::Google.Protobuf;

```

If you treat warnings as errors, then the compiler complains and refuses to compile the source code, as shown in the following output:

```
Error CS8981 The type name 'pb' only contains
lower-cased ascii characters. Such names may
become reserved for the language.
Northwind.Grpc.Service C:\apps-services-
net10\Chapter14\Northwind.Grpc.Service\obj
\Debug\net10.0\Protos\Greet.cs
```

Good Practice: Always treat warnings as errors in your .NET projects (except for gRPC projects until Google updates their code generation tools).

If you find that you get too many errors after enabling this, you can disable specific warnings by using the `<WarningsNotAsErrors>` element with a comma-separated list of warning codes, as shown in the following markup:

```
<WarningsNotAsErrors>0219, CS8981</
WarningsNotAsErrors>
```

The warning codes may include their prefix or not, for example, CS8981 or 8991.

You can learn more about controlling warnings as errors at the following link: <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/compiler-options/errors-warnings#warningsaserrors-and-warningsnotaserrors>.

Now that you have set up your development environment and configured central package management for your projects, let's build an entity model for use in the rest of the book to provide data for the apps that you will build.

Building an entity model for use in the rest of the book

Apps, websites, and web services usually need to work with data in a relational database or another data store. There are several technologies that could be used, from lower-level ADO.NET to higher-level EF Core, but we will use EF Core since it is flexible and more familiar to .NET developers.

In this section, we will define an EF Core entity data model for a database named Northwind stored in SQL Server. It will be used in most of the projects that we create in subsequent chapters. You will learn more details about SQL Server and EF Core in [Chapter 7, Managing Relational Data Using SQL](#) and [Chapter 8, Building Entity Models Using EF Core](#).

Northwind database SQL scripts

The script for SQL Server creates 13 tables as well as related views and stored procedures. The SQL scripts are found at the following link:

<https://github.com/markjprice/apps-services-net10/tree/main/scripts/sql-scripts>.

I recommend that in your `apps-services-net10` folder, you create a `sql-scripts` folder and copy all the SQL scripts to that local folder.

There are multiple SQL scripts to choose from, as described in the following list:

- `Northwind4SqlServerContainer.sql` script: To use SQL Server on a local computer in a container system like Docker. The script creates the Northwind database. It does not drop the database if it already exists because the Docker container should be empty anyway as a fresh one can be spun up each time. Instructions to install Docker and set up a SQL Server image and container are in the next section of this book. This is my recommendation for using SQL Server in this book.

- `Northwind4SqlServerLocal.sql` script: To use SQL Server on a local Windows or Linux computer. The script checks if the Northwind database already exists and if necessary, drops (deletes) it before creating it. Instructions to install SQL Server Developer Edition (free) on your local Windows computer can be found in the GitHub repository for this book at the following link: <https://github.com/markjprice/apps-services-net10/blob/main/docs/sql-server/README.md>.
- `Northwind4SqlServerCloud.sql` script: To use SQL Server with an Azure SQL Database resource created in the Azure cloud. You will need an Azure account; these resources cost money as long as they exist! The script does not drop or create the Northwind database because you should manually create the Northwind database using the Azure portal user interface. The script only creates the database objects, including the table structure and data.

Before you can execute any of these SQL scripts, you need a SQL Server instance. My recommendation is to use Docker and a container, so that's what we will cover in the next section. If you prefer a local or cloud SQL Server, then you can skip this next section.

Installing Docker and an SQL Server container image

Docker provides a consistent environment across development, testing, and production, minimizing the “it works on my machine” issue. Docker containers are more lightweight than traditional virtual machines, making them faster to start up and less resource-intensive.

Docker containers can run on any system with Docker installed, making it easy to move databases between environments or across different machines. You can quickly spin up a SQL database container with a single command, making setup faster and more reproducible. Each database instance runs in its own container,

ensuring that it is isolated from other applications and databases on the same machine.

You can install Docker on any operating system and use a container that has SQL Server installed. For personal, educational, and small business use, Docker Desktop is free to use. It includes the full set of Docker features, including container management and orchestration. The Docker **Command-line Interface (CLI)** and Docker Engine are open source and free to use, allowing developers to build, run, and manage containers.

Docker also has paid tiers that offer additional features, such as enhanced security, collaboration tools, more granular access control, priority support, and higher rate limits on Docker Hub image pulls.

The Docker image we will use has SQL Server 2025 hosted on Ubuntu 22.04. It is supported with Docker Engine 1.8 or later.

Let’s install Docker and set up the SQL image and container now:

1. Install Docker Desktop from the following link: <https://docs.docker.com/engine/install/>.
2. Start **Docker Desktop**, which could take a few minutes on the initial start, as shown in *Figure 1.5*:

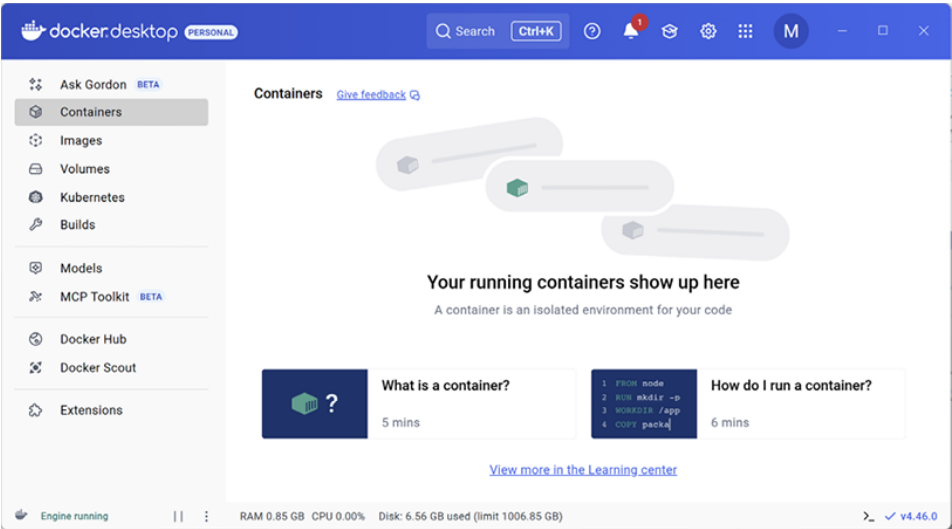


Figure 1.5: Docker Desktop v4.46 (September 2025) on Windows

1. At the command prompt or terminal, pull down the latest container image for SQL Server 2025, as shown in the following command:

```
docker pull mcr.microsoft.com/mssql/  
server:2025-latest
```

Unfortunately, the recent SQL Server images from Microsoft only support x64 architecture. If you want to use an image that runs without emulation on ARM CPUs, for example, if you have a Surface Laptop 7 or Mac, then you can use a minimal edition of SQL Server known as Azure SQL Edge that runs on either x64 or ARM64, with a minimum of 1 GB RAM on the host. But Azure SQL Edge is no longer supported by Microsoft, so use it at your own risk. I think that it's fine for learning purposes, but do not use unsupported software in production. To pull the Azure SQL Edge image, enter the following command: `docker pull mcr.microsoft.com/azure-sql-edge:latest`

1. Wait for the image as it is downloading, and then note the results, as shown in the following output:

```
2025-latest: Pulling from mssql/server  
a7f551132cc7: Pull complete d39c64e0c073:
```

```
Pull complete 04a0776f5c78: Pull complete  
Digest:  
sha256:e2e5bcfe395924ff49694542191d3aefe86b6b3bd6  
Status: Downloaded newer image for  
mcr.microsoft.com/mssql/server:2025-latest
```

Running the SQL Server container image

You can create a container from the image and run it in a single `docker run` command with the following options:

- `--cap-add SYS_PTRACE`: This grants the container the `SYS_PTRACE` capability allows debugging tools (like `strace` or certain profilers) to attach to processes within the container. Microsoft recommends this for enabling debugging or diagnostic tools inside the container, but it's not strictly necessary for normal SQL Server operation.
- `-e 'ACCEPT_EULA=1'` or `-e "ACCEPT_EULA=1"`: Sets an environment variable to accept the SQL Server End User License Agreement. If you don't provide this, the container will exit immediately with a message saying you must accept the EULA.
- `-e 'MSSQL_SA_PASSWORD=s3cret-Ninja'` or `-e "MSSQL_SA_PASSWORD=s3cret-Ninja"`: Sets an environment variable to set the SQL Server `sa` (system administrator) account's password. The password must be at least eight characters long and contain characters from three of the following four sets: uppercase letters, lowercase letters, digits, and symbols. Otherwise, the container cannot set up the SQL Server engine and will fail. `s3cret-Ninja` satisfies those rules (lowercase, number, hyphen, uppercase) but feel free to use your own password if you wish.
- `-p 1433:1433`: This maps a port from the container to the host. 1433 is the default port for SQL Server. This allows applications on the host to connect to SQL Server inside the container as if it were running natively.

- `--name nw-container`: Gives the container a custom name. This is optional, but if you don't set a name, a random one will be assigned for you, like `frosty_mirzakhani`.
- `-d`: Runs the container in detached mode (in the background). Without it, Docker would run the container in the foreground, tying up your terminal or command prompt window.
- `mcr.microsoft.com/mssql/server:2025-latest`: Specifies the image to run. `mcr.microsoft.com` is Microsoft's official container registry. `mssql/server` is the SQL Server on Linux container image. `2025-latest` is the tag for the latest build of SQL Server 2025.

Now that you understand what you are about to do, we can run the image:

1. At the command prompt or terminal, run the container image for SQL Server with a strong password and name the container `nw-container`, as shown in the following command:

```
docker run --cap-add SYS_PTRACE -e
'ACCEPT_EULA=1' -e
'MSSQL_SA_PASSWORD=s3cret-Ninja' -p
1433:1433 --name nw-container -d
mcr.microsoft.com/mssql/server:2025-latest
```

Warning! The preceding command must be entered all on one line, or the container will not work correctly. In particular, the container might start up but without a password set and therefore later you won't be able to connect to it! All command lines used in this book can be found and copied from the following link:

<https://github.com/markjprice/apps-services-net10/blob/main/docs/command-lines.md>. Also, different operating systems may require different quote characters, or none at all. To set the environment variables you should be able to use either straight single-quotes or straight double-quotes. If the logs show that you did not accept the license agreement, then try the other type of quotes.

With Windows 11 on my Surface Laptop 7 (ARM64), running the container image at the command prompt failed for me. See the next section titled *Running a container using the user interface* for steps that worked.

1. If your operating system firewall blocks access, then allow access.
2. In **Docker Desktop**, in the **Containers** section, confirm that the image is running, as shown in *Figure 1.6*:

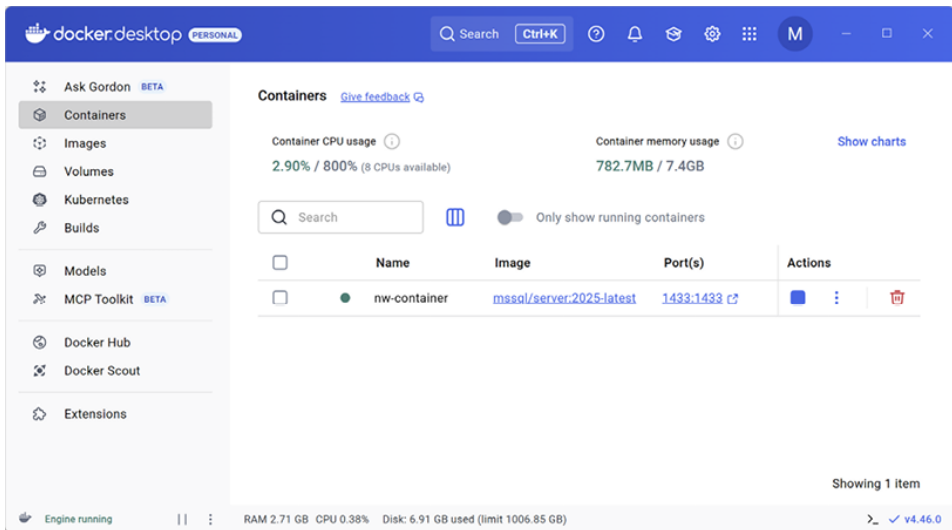


Figure 1.6: SQL Server container running in Docker Desktop on Windows

You might assume that the link in the **Port(s)** column is clickable and will navigate to a working website. But that container image only has SQL Server in it. SQL Server is listening on that port and can be connected to using a TCP address, not an HTTP address, so Docker is misleading you! There is no web server listening on port 1433 so a web browser that makes a request to <http://localhost:1433> will get a **This page isn't working** error.

This is expected behavior because a database server is not a web server. Many containers in Docker do host a web server, and in those scenarios having a convenient clickable link is useful. But Docker has no idea which containers have web servers and which do not. All it knows is what ports are mapped from internal ports to external ports. It is up to the developer to know if those links are useful.

1. At the command prompt or terminal, ask Docker to list all containers, both running and stopped, as shown in the following command:

```
docker ps -a
```

2. Note the container STATUS is “Up 53 seconds” and it is listening externally on port 1433, which is mapped to its internal port 1433, as shown highlighted in the following output:

```
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES 183f02e84b2a
mcr.microsoft.com/ mssql/server:2025-
latest "/opt/mssql/bin/perm..." 8 minutes
ago Up 53 seconds 1401/tcp, 0.0.0.0:1433-
>1433/tcp nw-container
```

You can learn more about the `docker ps` command at <https://docs.docker.com/engine/reference/commandline/ps/>.

Running a container using the user interface

If you successfully ran the SQL Server container, then you can skip this section and continue with the next section, titled *Connecting to SQL Server in a Docker container*.

If entering a command at the prompt or terminal fails for you, try following these steps to use the user interface:

1. In **Docker Desktop**, navigate to the **Images** tab.
2. In the **mcr.microsoft.com/mssql/server** row, in the **Actions** column, click the **Run** button.
3. In the **Run a new container** dialog box, expand **Optional settings**, and complete the configuration, as shown in *Figure 1.7* and in the following items:

- **Container name:** `nw-container`, or leave blank to

use a random name.

- **Ports:** Enter 1433 to map to :1433/tcp.
- **Volumes:** leave empty.
- **Environment variables** (click + to add a second one):
 - Enter ACCEPT_EULA with value Y (or 1).
 - Enter MSSQL_SA_PASSWORD with value s3cret-Ninja.

10. Click **Run**.

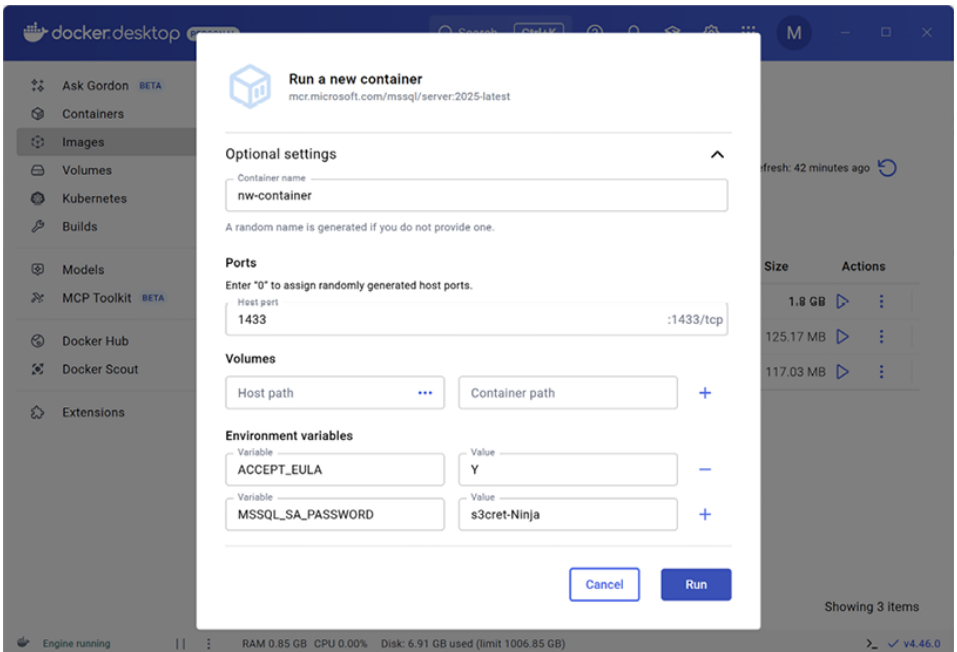


Figure 1.7: Running a container for SQL Server with the user interface

Connecting to SQL Server in a Docker container

Use your preferred database tool to connect to SQL Server in the Docker container. Some common database tools are shown in the following list:

- Windows only:
 - **SQL Server Management Studio (SSMS)**: The most popular and comprehensive tool for managing SQL Server databases. Free to download from Microsoft.
 - **SQL Server Data Tools (SSDT)**: Integrated into Visual Studio and free to use, SSDT provides database development tools for designing, deploying, and managing SQL Server databases.
- Cross-platform for Windows, macOS, Linux:
 - **VS Code's MS SQL extension**: Query execution, IntelliSense, database browsing, and connection to SQL Server databases.

Some notes about the database connection string for SQL Server in a container:

- **Data Source**, a.k.a. **Server Name**: `tcp:127.0.0.1,1433`
- **Authentication**: You must use **SQL Server Authentication**, a.k.a. **SQL Login**. That is, you must supply a username and password. The SQL Server image has the `sa` user already created, and you had to give it a strong password when you ran the container. We chose the password `s3cret-Ninja`.
- You must select the **Trust Server Certificate** check box.
- Optionally, you might want to save your password for future use.
- **Initial Catalog**, a.k.a. **database**: `master` or leave blank. (We will create the Northwind database using a SQL script so we do not specify that as the database name yet.)

Warning! If you already have SQL Server installed locally, and its services are running, then it will be listening to port 1433, and it will take priority over any Docker-hosted SQL Server services that are also trying to listen on port 1433. You will need to stop the local SQL Server before being able to

connect to any Docker-hosted SQL Server services. You can do this using Windows **Services**: in the **Services (Local)** list, right-click **SQL Server (MSSQLSERVER)** and choose **Stop**. (This can take a few minutes so be patient.) You can also right-click and choose **Properties** and then set the **Startup type** to **Manual**, as shown in *Figure 1.8* (it defaults to **Automatic**, so if you restart Windows, it will be running again). Or change the port number(s) for either the local or Docker-hosted SQL Server services so that they do not conflict.

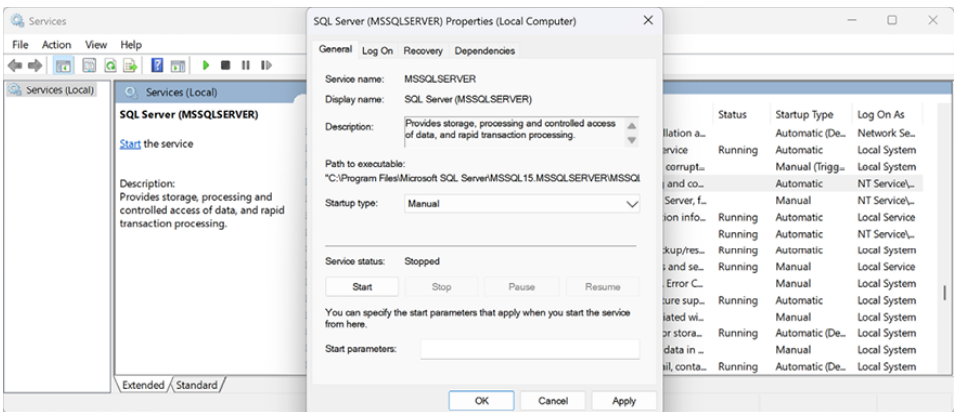


Figure 1.8: SQL Server service properties

I have created a troubleshooting guide if you have trouble connecting: <https://github.com/markjprice/apps-services-net10/blob/main/docs/errata/sql-container-issues.md>.

Connecting from Visual Studio

To connect to SQL Server using Visual Studio:

1. In Visual Studio, navigate to **View | Server Explorer**.
2. In the **Server Explorer** mini-toolbar, click the **Connect to Database...** button.
3. If prompted to **Change Data Source**, then choose **Microsoft SQL Server**.
4. Enter the connection details, as shown in *Figure 1.9*:

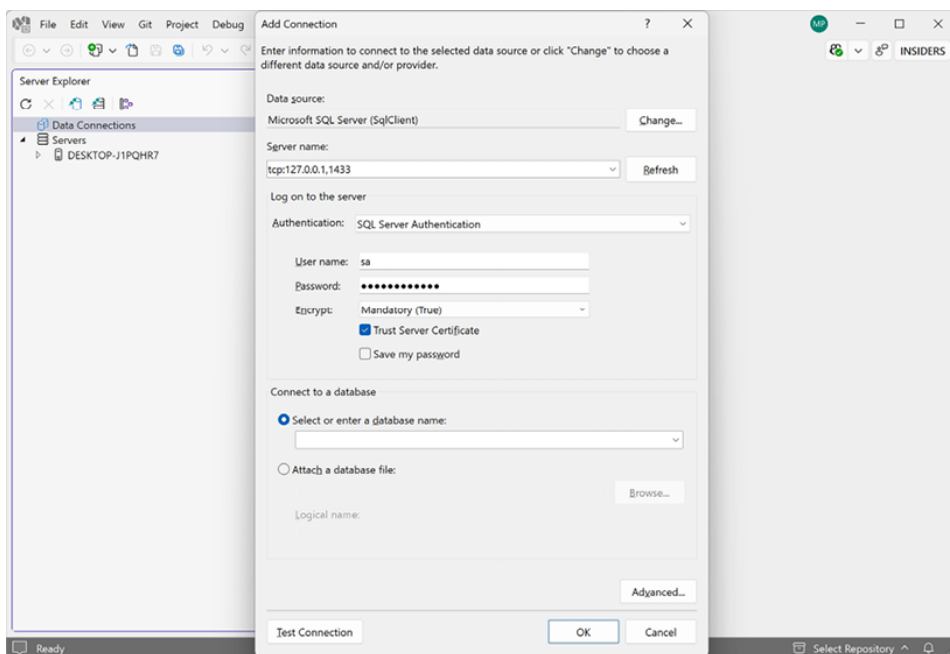


Figure 1.9: Connecting to your SQL Server in a container from Visual Studio

Warning! If you get the error: **Login failed for user 'sa'**, then the most likely causes are either that the password was not set correctly when you ran the Docker container, or you are connecting to a different SQL Server, for example, a local one instead of the one in the container.

Connecting from VS Code

To connect to SQL Server in a container using VS Code:

1. In VS Code, navigate to the **SQL Server** extension. Note that the `mssql` extension might take a few minutes to initialize the first time.
2. In the **SQL** extension, click **Add Connection....**
3. In the **Connect to Database** pane, enter the connection details, as shown in *Figure 1.10*:

- **Profile Name:** `SQL Server in Container`
- **Connection Group:** `<Default>`
- **Input type:** `Parameters`
- **Server name:** `tcp:127.0.0.1,1433`
- **Trust Server Certificate:** `Selected`
- **Authentication type:** `SQL Login`
- **User name:** `sa`
- **Password:** `s3cret-Ninja`
- **Save Password:** `Cleared` (or selected for convenience during learning)
- **Database name:** `master` or leave blank. (We will create the Northwind database using a SQL script so we do not specify that as the database name yet.)
- **Encrypt:** `Mandatory`

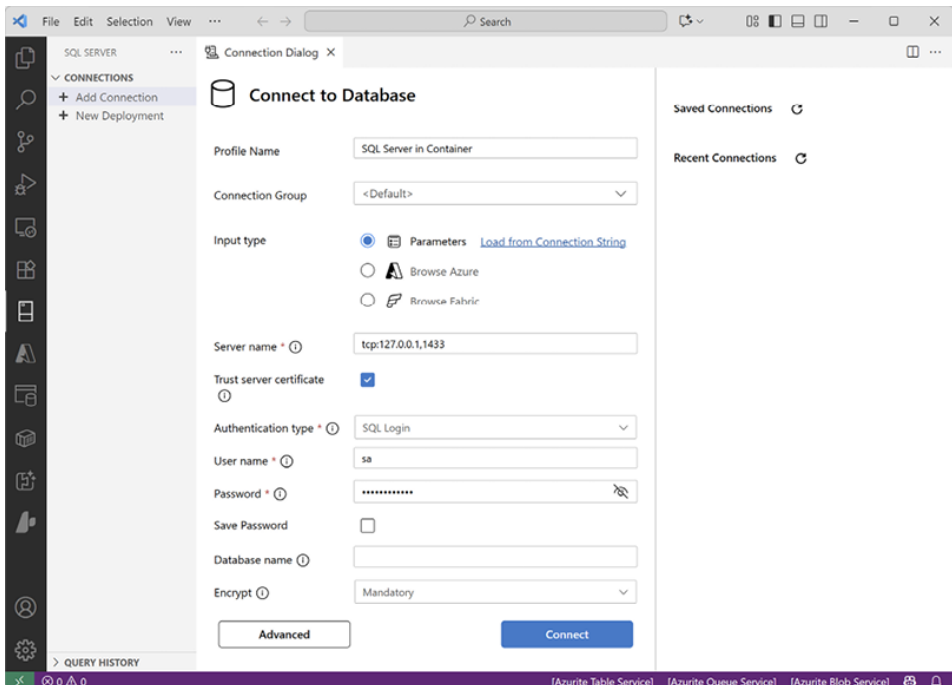


Figure 1.10: Connecting to your SQL Server in a container from VS Code

1. Note the success notification message.

Creating the Northwind database using a SQL script

Now you can use your preferred code editor (or database tool) to execute the SQL script to create the Northwind database in SQL Server in a container:

1. Open the `Northwind4SqlServerContainer.sql` file. Note that this file does not know about the **Server Explorer** connection to the SQL Server database.
2. Connect the file to the SQL Server database. For example, in Visual Studio, right-click in the script file, navigate to **Connection | Connect...**, and then fill in the dialog box as before.
3. Execute the SQL script:

- If you are using Visual Studio, right-click in the script file, select **Execute**, and then wait to see the Command completed successfully message.
- If you are using VS Code, right-click in the script file, select **Execute Query**, select the **SQL Server in a Container** connection profile, and then wait to see the Commands completed successfully message.

6. Refresh the data connection:

- If you are using Visual Studio, then in **Server Explorer**, right-click **Tables** and select **Refresh**.
- If you are using VS Code, then right-click the **SQL Server in a Container** connection profile and choose **Refresh**.

9. Expand **Databases**, expand **Northwind**, and then expand **Tables**.

10. Note that 13 tables have been created, for example, **Categories**, **Customers**, and **Products**. Also note that dozens of views and stored procedures have also been created, as shown in *Figure 1.11*:

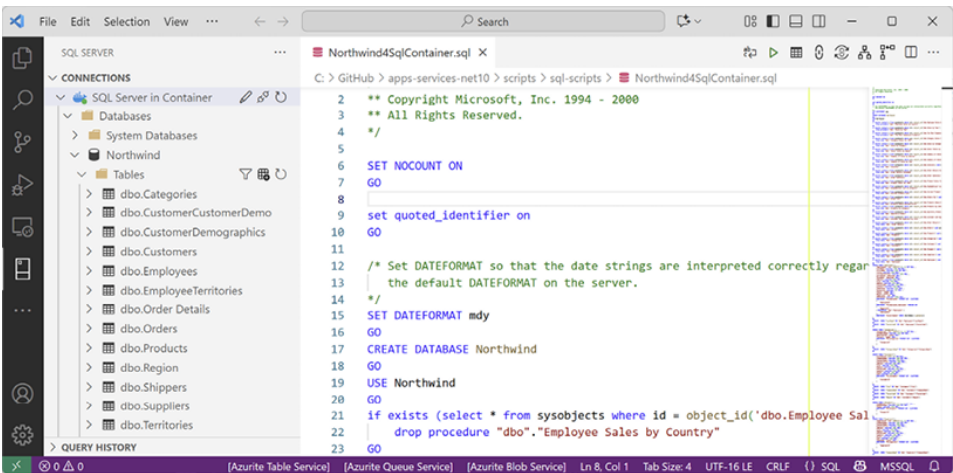


Figure 1.11: Northwind database created by SQL script in VS Code

You now have a running instance of SQL Server containing the Northwind

database that you can connect to from your ASP.NET Core projects.

You will want to keep the container while you work through all the chapters in this book. You can stop and start the container whenever you want and the database will persist. Eventually, once you have finished this book, you might want to delete the container. This will also delete the database, so if you recreate the container, you will need to rerun the SQL script to recreate the Northwind database.

Removing Docker resources

When you have completed all the chapters in the book, or you plan to use a local SQL Server or Azure SQL Database in the cloud instead of a SQL Server container, and you want to remove all the Docker resources that it uses, then either use the Docker Desktop user interface, or follow these steps at the command prompt or terminal:

1. At the command prompt or terminal, stop the `nw-container` container, as shown in the following command:

```
docker stop nw-container
```

2. At the command prompt or terminal, remove the `nw-container` container, as shown in the following command:

```
docker rm nw-container
```

Warning! Removing the container will delete all data inside it.

1. At the command prompt or terminal, remove the image to release its disk space, as shown in the following command:

```
docker rmi mcr.microsoft.com/mssql/  
server:2025-latest
```

Setting up the EF Core CLI tool

The .NET CLI tool named `dotnet` can be extended with capabilities useful for working with EF Core. It can perform design-time tasks like creating and applying migrations from an older model to a newer model and generating code for a model from an existing database.

The `dotnet-ef` command-line tool is not automatically installed. You must install this package as either a **global** or **local tool**. If you have already installed an older version of the tool, then you should update it to the latest version:

1. At a command prompt or terminal, check if you have already installed `dotnet-ef` as a global tool, as shown in the following command:

```
dotnet tool list --global
```

2. Check in the list if an older version of the tool has been installed, like the one for .NET 9, as shown in the following output:

```
Package Id Version Commands  
-----  
dotnet-ef 9.0.0 dotnet-ef
```

3. If an old version is installed, then update the tool, as shown in the following command:

```
dotnet tool update --global dotnet-ef
```


4. If it is not already installed, then install the latest version, as shown in the following command:

```
dotnet tool install --global dotnet-ef
```

If necessary, follow any OS-specific instructions to add the `dotnet tools` directory to your `PATH` environment variable, as described in the output of installing the `dotnet-ef` tool.

By default, the latest GA release of .NET will be used to install the tool. To explicitly set a version, for example, to use a preview, add the `--version` switch. For example, to update to the latest .NET 11 preview or release candidate version (which will be available from February 2026 to October 2026), use the following command with a version wildcard:

```
dotnet tool update --global dotnet-ef --  
version 11.0-*
```

Once the .NET 11 GA release happens in November 2026, you can just use the command without the `--version` switch to upgrade.

You can also remove the tool, as shown in the following command:

```
dotnet tool uninstall --global dotnet-ef
```

Creating a class library for entity models

You will now define entity data models in a class library so that they can be reused in other types of projects, including client-side app models.

Good practice: You should create a separate class library project for your entity data models from the

class library for your database context. This allows easier sharing of the entity models between backend web servers and frontend desktop, mobile, and Blazor clients, while only the backend needs to reference the database context class library.

We will automatically generate some entity models using the EF Core command-line tool:

1. Use your preferred code editor to create a new project and solution, as defined in the following list:
 - Project template: **Class Library** / `classlib`
 - Project file and folder: `Northwind.EntityModels`
 - Solution file and folder: `ModernApps`
5. In the `Northwind.EntityModels.csproj` project file, add package references for the SQL Server database provider and EF Core design-time support, as shown in the following markup:

```
<ItemGroup> <PackageReference
  Include="Microsoft.EntityFrameworkCore.SqlServer"
/> <PackageReference
  Include="Microsoft.EntityFrameworkCore.Design">
  <PrivateAssets>all</PrivateAssets>
  <IncludeAssets>runtime; build; native;
  contentfiles; analyzers; buildtransitive</
  IncludeAssets> </PackageReference> </
ItemGroup>
```

6. Delete the `Class1.cs` file.
7. Build the `Northwind.EntityModels` project to restore packages.
8. Make sure that the SQL Server container is running because you are about to connect to the server and its Northwind database.

9. At a command prompt or terminal, in the Northwind.EntityModels project folder (the folder that contains the .csproj project file), generate entity class models for all tables, as shown in the following command:

```
dotnet ef dbcontext scaffold "Data
Source=tcp:127.0.0.1,1433;Initial
Catalog=Northwind;User
Id=sa;Password=s3cret-
Ninja;TrustServerCertificate=true;"
Microsoft.EntityFrameworkCore.SqlServer --
namespace Northwind.EntityModels --data-
annotations
```

Note the following:

- The command to perform: `dbcontext scaffold`
- The connection string: `"Data Source=tcp:127.0.0.1,1433;Initial Catalog=Northwind;User Id=sa;Password=s3cret-Ninja;TrustServerCertificate=true;"`
- The database provider: `Microsoft.EntityFrameworkCore.SqlServer`
- The namespace: `--namespace Northwind.EntityModels`
- To use data annotations as well as the Fluent API: `--data-annotations`

Warning! `dotnet-ef` commands must be entered all on one line and in a folder that contains a project, or you will see

the following error: No project was found. Change the current working directory or use the --project option. Remember that all command lines can be found at and copied from the following link: <https://github.com/markjprice/apps-services-net10/blob/main/docs/command-lines.md>.

If you are using a local instance of SQL Server then you can use the following command:

```
dotnet ef dbcontext scaffold "Data
Source=.;Initial Catalog=Northwind;Integrated
Security=true;TrustServerCertificate=true;"
Microsoft.EntityFrameworkCore.SqlServer --
namespace Northwind.EntityModels --data-
annotations
```

Note the different data source and authentication in the connection string:

```
"Data Source=.;Initial Catalog=Northwind;Integrated
Security=true;TrustServerCertificate=true;"
```

Creating a class library for a database context

You will now define a database context class library:

1. Add a new project to the solution, as defined in the following list:
 - Project template: **Class Library** / classlib
 - Project file and folder: Northwind.DataContext
 - Solution file and folder: ModernApps
5. In the Northwind.DataContext project, statically and globally import the Console class, add a package reference to the EF Core data provider for SQL Server, and add a project reference to the Northwind.EntityModels project, as shown in the following markup:

```
<ItemGroup Label="To simplify use of
WriteLine."> <Using
Include="System.Console" Static="true" />
</ItemGroup> <ItemGroup Label="Versions
are set at solution-level.">
<PackageReference
Include="Microsoft.EntityFrameworkCore.SqlServer"
/> </ItemGroup> <ItemGroup>
<ProjectReference Include="..
\Northwind.EntityModels
\Northwind.EntityModels.csproj" /> </
ItemGroup>
```

Warning! The path to the project reference should not have a line break in your project file.

1. In the Northwind.DataContext project, delete the Class1.cs file.
2. Build the Northwind.DataContext project to restore packages.
3. In the Northwind.DataContext project, add a class named NorthwindContextLogger.cs.

4. Modify its contents to define a static method named `WriteLine` that appends a string to the end of a text file named `northwindlog-<date_time>.txt` on the desktop, as shown in the following code:

```
using static System.Environment; namespace
Northwind.EntityModels; public class
NorthwindContextLogger { public static
void WriteLine(string message) { string
folder = Path.Combine(GetFolderPath(
SpecialFolder.DesktopDirectory), "book-
logs"); if (!Directory.Exists(folder))
Directory.CreateDirectory(folder); string
dateTimeStamp =
DateTime.Now.ToString("yyyyMMdd_HH:mm:ss");
string path = Path.Combine(folder,
$"northwindlog-{dateTimeStamp}.txt");
StreamWriter textFile =
File.AppendText(path);
textFile.WriteLine(message);
textFile.Close(); } }
```

Although the project name (and therefore default assembly name) is `Northwind.DataContext`, to simplify usage of the data context class we have defined it in the same namespace as the related models: `Northwind.EntityModels`.

1. Move the `NorthwindContext.cs` file from the `Northwind.EntityModels` project/folder to the `Northwind.DataContext` project/folder.

Warning! In Visual Studio **Solution Explorer**, if you drag and drop a file between projects, it will be copied. If you hold down *Shift* while dragging and dropping, it will be moved. In VS Code **EXPLORER**, if you drag and drop a file between projects, it will be moved. If you hold down *Ctrl* while dragging and dropping, it will be copied.

1. In `NorthwindContext.cs`, note the second constructor can have options passed as a parameter, which allows us to override the default database connection string in any projects, such as websites that need to work with the Northwind database, as shown in the following code:

```
public NorthwindContext(  
    DbContextOptions<NorthwindContext>  
    options) : base(options) { }
```

2. In `NorthwindContext.cs`, both constructors give dozens of warnings, one for each of its `DbSet<T>` properties that represent tables and views, because they are marked as not nullable, and the compiler does not know that EF Core will automatically instantiate them all, so they will never actually be null. We can hide these warnings by disabling warning code CS8618 just for those two constructors, as shown in the following code:

```
public partial class NorthwindContext :  
    DbContext { #pragma warning disable CS8618  
    public NorthwindContext() #pragma warning  
        restore CS8618 { } #pragma warning disable  
        CS8618 public
```

```
NorthwindContext (DbContextOptions<NorthwindContext>
options) #pragma warning restore CS8618 :
base(options) { }
```

Good practice: We could simplify the code by disabling that warning code once at the top of the file and not restoring it anywhere in that file, but it is safer to re-enable warning codes in case you encounter more instances that you do need to handle differently.

You can learn more about this warning code at the following link: <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/compiler-messages/nullable-warnings#nonnullable-reference-not-initialized>.

1. In `NorthwindContext.cs`, in the `OnConfiguring` method, remove the compiler #warning about the connection string and then add statements to dynamically build a database connection string for SQL Server in a container, as shown in the following code:

```
protected override void OnConfiguring(
DbContextOptionsBuilder optionsBuilder) {
    if (!optionsBuilder.IsConfigured) {
        SqlConnectionStringBuilder builder =
        new(); builder.DataSource =
        "tcp:127.0.0.1,1433"; // SQL Server in
```



```

container. builder.InitialCatalog =
"Northwind";
builder.TrustServerCertificate = true;
builder.MultipleActiveResultSets = true;
// Because we want to fail faster. Default
is 15 seconds. builder.ConnectTimeout = 3;
// SQL Server authentication.
builder.UserID = Environment
.GetEnvironmentVariable("MY_SQL_USR");
builder.Password = Environment
.GetEnvironmentVariable("MY_SQL_PWD");
optionsBuilder.UseSqlServer(builder.ConnectionString);
optionsBuilder.LogTo(NorthwindContextLogger.WriteLine);
new[] { Microsoft.EntityFrameworkCore
.Diagnostics.RelationalEventId.CommandExecuting
}); } }

```

2. In the `Northwind.DataContext` project, add a class named `NorthwindContextExtensions.cs`. Modify its contents to define an extension method that adds the Northwind database context to a collection of dependency services, as shown in the following code:

```

using Microsoft.Data.SqlClient; // To use
SqlConnectionStringBuilder. using
Microsoft.EntityFrameworkCore; // To use
UseSqlServer. using
Microsoft.Extensions.DependencyInjection;
// To use IServiceCollection. namespace
Northwind.EntityModels; public static
class NorthwindContextExtensions { ///
<summary> /// Adds NorthwindContext to the
specified IServiceCollection. Uses the
SqlServer database provider. /// </
summary> /// <param name="services">The

```

```
service collection.</param> /// <param
name="connectionString">Set to override
the default.</param> /// <returns>An
IServiceCollection that can be used to add
more services.</returns> public static
IServiceCollection AddNorthwindContext(
this IServiceCollection services, // The
type to extend. string? connectionString =
null) { if (connectionString is null) {
SqlConnectionStringBuilder builder =
new(); builder.DataSource =
"tcp:127.0.0.1,1433"; // SQL Server in
container. builder.InitialCatalog =
"Northwind";
builder.TrustServerCertificate = true;
builder.MultipleActiveResultSets = true;
// Because we want to fail faster. Default
is 15 seconds. builder.ConnectTimeout = 3;
// SQL Server authentication.
builder.UserID =
Environment.GetEnvironmentVariable("MY_SQL_USR");
builder.Password =
Environment.GetEnvironmentVariable("MY_SQL_PWD");
connectionString =
builder.ConnectionString; }
services.AddDbContext<NorthwindContext>(options
=> {
options.UseSqlServer(connectionString);
options.LogTo(NorthwindContextLogger.WriteLine,
new[] { Microsoft.EntityFrameworkCore
.Diagnostics.RelationalEventId.CommandExecuting
}); }, // Register with a transient
lifetime to avoid concurrency // issues
with Blazor Server projects.
contextLifetime:
```

```
ServiceLifetime.Transient,  
optionsLifetime:  
ServiceLifetime.Transient); return  
services; } }
```

3. Build the two class libraries and fix any compiler errors.

There is duplicate code in these two classes because the `NorthwindContext` class and its extensions are written to allow developers to instantiate the context class directly as well as via the extension method. They can also override the connection string or choose to accept defaults.

Setting the user and password for SQL Server authentication

If you are using SQL Server authentication, i.e., you must supply a user and password, then complete the following steps:

1. In the `Northwind.DataContext` project, note the statements that set `UserId` and `Password`, as shown in the following code:

```
// SQL Server authentication.  
builder.UserId = Environment  
.GetEnvironmentVariable("MY_SQL_USR");  
builder.Password = Environment  
.GetEnvironmentVariable("MY_SQL_PWD");
```

2. Set the two environment variables at the command prompt or terminal, as shown in the following commands:
 - On Windows:

```
setx MY_SQL_USR <your_user_name> setx  
MY_SQL_PWD <your_password>
```

- On macOS and Linux:

```
export MY_SQL_USR=<your_user_name> export  
MY_SQL_PWD=<your_password>
```

Unless you chose a different password, `<your_user_name>` will be `sa`, and `<your_password>` will be `s3cret-Ninja`.

1. You **must** restart any command prompts, terminal windows, and applications like Visual Studio for this change to take effect.

Good practice: Although you could define the two environment variables in the `launchSettings.json` file of an ASP.NET Core project, you must then be extremely careful not to include that file in a GitHub repository! You can learn how to ignore files in Git at <https://docs.github.com/en/get-started/getting-started-with-git/ignoring-files>.

Improving the class-to-table mapping

We will make some small changes to improve the entity model mapping and validation rules for SQL Server.

Remember that all code is available in the GitHub repository for the book. Although you will learn more by typing the code yourself, you never have to. Go to the following link and press . to get a live code editor in your browser: <https://github.com/markjprice/apps-services-net10>.

We will add a regular expression to validate that a `CustomerId` value is exactly five uppercase letters:

1. In `Customer.cs`, add a regular expression to validate its primary key `CustomerId` to only allow uppercase Western characters, as shown highlighted in the following code:

```
[Key] [StringLength(5)]  
[RegularExpression("[A-Z]{5}")] public  
string CustomerId { get; set; } = null!;
```

2. In `Customer.cs`, add the `[Phone]` attribute to its `Phone` property, as shown highlighted in the following code:

```
[StringLength(24)] [Phone] public string?  
Phone { get; set; }
```

The `[Phone]` attribute adds the following to the rendered HTML: `type="tel"`. On a mobile phone, this makes the keyboard use the phone dialer instead of the normal keyboard.

1. In `Order.cs`, decorate the `CustomerId` property with the same

regular expression to enforce five uppercase characters.

Testing the class libraries using xUnit

xUnit is a popular unit testing framework for .NET applications. It was created by the original inventor of NUnit and is designed to be more modern, extensible, and aligned with .NET development practices.

Several benefits of using xUnit are shown in the following list:

- xUnit is open-source and has a strong community and active development team behind it. This makes it more likely that it will stay up to date with the latest .NET features and best practices. xUnit benefits from a large and active community, which means many tutorials, guides, and third-party extensions are available for it.
- xUnit uses a more simplified and extensible approach compared to older frameworks. It encourages the use of custom test patterns and less reliance on setup and teardown methods, leading to cleaner test code.
- Tests in xUnit are configured using .NET attributes, which makes the test code easy to read and understand. It uses `[Fact]` for standard test cases and `[Theory]` with `[InlineData]`, `[ClassData]`, or `[MemberData]` for parameterized tests, enabling data-driven testing. This makes it easier to cover many input scenarios with the same test method, enhancing test thoroughness while minimizing effort.
- xUnit includes an assertion library that allows for a wide variety of assertions out of the box, making it easier to test a wide range of conditions without having to write custom test code. It can also be extended with popular assertion libraries, like `FluentAssertions`, that allow you to articulate test expectations with human-readable reasons.
- By default, xUnit supports parallel test execution within the same test collection, which can significantly reduce the time it takes to run large test suites. This is particularly beneficial in continuous integration

environments where speed is critical. However, if you run your tests in a memory-limited **VPS (Virtual Private Server)**, then that impacts how much data the server can handle at any given time and how many applications or processes it can run concurrently. In this scenario, you might want to disable parallel test execution. Memory-limited VPS instances are typically used as cheap testing environments.

- xUnit offers precise control over the test lifecycle with setup and teardown commands through the use of the constructor and destructor patterns and the `IDisposable` interface, as well as with the `[BeforeAfterTestAttribute]` for more granular control.

Now let's build some unit tests to ensure the class libraries are working correctly:

1. Use your preferred coding tool to add a new **xUnit Test Project [C#]** / xunit project named `Northwind.UnitTests` to the `ModernApps` solution.
2. In the `Northwind.UnitTests` project, make changes as described in the following bullets, and as shown in the following configuration:
 - Delete the version numbers specified for the testing packages in the project file. (Visual Studio and other code editors will give errors if you have projects that should use CPM but specify their own package versions without using the `VersionOverride` attribute.)
 - Add a project reference to the `Northwind.DataContext` project:

```
<ItemGroup> <PackageReference
Include="coverlet.collector" />
<PackageReference
Include="Microsoft.NET.Test.Sdk"
/> <PackageReference
Include="xunit" />
```

```

<PackageReference
Include="xunit.runner.visualstudio"
/> </ItemGroup> <ItemGroup>
<ProjectReference Include="..
\Northwind.DataContext
\Northwind.DataContext.csproj"
/> </ItemGroup>

```

Warning! The project reference must go all on one line with no line break.

1. Build the Northwind.UnitTests project to build referenced projects.
2. Rename UnitTest1.cs to EntityModelTests.cs.
3. Modify the contents of the file to define two tests, the first to connect to the database and the second to confirm there are eight categories in the database, as shown in the following code:

```

using Northwind.EntityModels; // To use
NorthwindContext. namespace
Northwind.UnitTests; public class
EntityModelTests { [Fact] public void
DatabaseConnectTest() { using
NorthwindContext db = new();
Assert.True(db.Database.CanConnect()); }
[Fact] public void CategoryCountTest() {
using NorthwindContext db = new(); int
expected = 8; int actual =
db.Categories.Count();
Assert.Equal(expected, actual); } [Fact]
public void ProductId1IsChaiTest() { using
NorthwindContext db = new(); string

```



```

expected = "Chai"; Product? product =
db.Products.Find(keyValues: 1); string
actual = product?.ProductName ??
string.Empty; Assert.Equal(expected,
actual); } }

```

4. If your database server is not running, (for example, because you are hosting it in Docker, a virtual machine, or in the cloud), then make sure to start it.
5. Run the unit tests:
 - If you are using Visual Studio, then navigate to **Test | Run All Tests**, and then view the results in **Test Explorer**.
 - If you are using VS Code, then in the `Northwind.UnitTests` project's **TERMINAL** window, run the tests, as shown in the following command: `dotnet test`. Alternatively, use the **TESTING** window if you have installed C# Dev Kit.
8. Note that the results should indicate that three tests ran, and all passed, as shown in *Figure 1.12*:

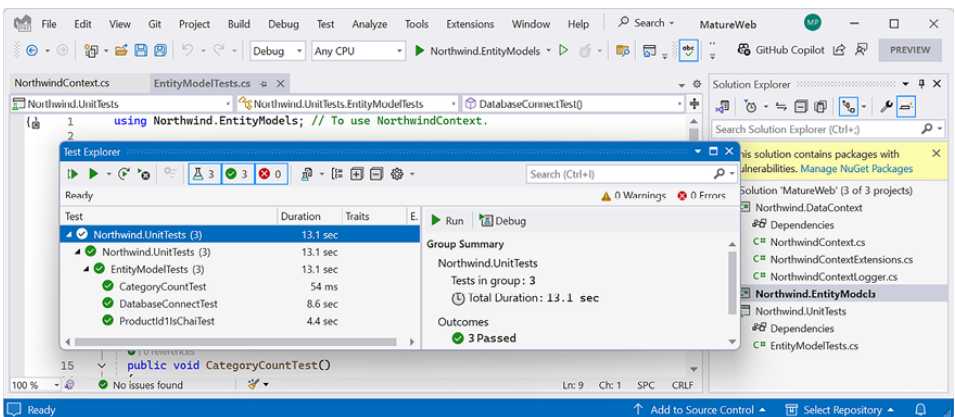


Figure 1.12: Three successful unit tests ran

If any of the tests fail, then try fix the issue.

For example, you might see the following exception:

```
System.ArgumentNullException : Value cannot be null. (Parameter 'User ID')
```

This occurs when the code tries to read the environment variable, but it has not been set. If you executed the commands to set the environment variables, then to fix the problem, restart Visual Studio and any terminals and command prompt windows. This will allow them to access the environment variables.

Now that we have built an entity model to use to work with sample data in all the projects in this book, let's end this chapter by looking at where you can get help when you get stuck.

Making good use of the GitHub repository for this book

Git is a commonly used source code management system. **GitHub** is a company, website, and desktop application that makes it easier to manage Git. Microsoft purchased GitHub in 2018, so it will continue to be closely integrated with Microsoft tools.

I created a GitHub repository for this book, and I use it for the following:

- To store the solution code for the book, which will be maintained after the print publication date.
- To provide extra materials that extend the book, like errata fixes, small improvements, lists of useful links, and longer articles that cannot fit in the printed book.
- To provide a place for readers to get in touch with me if they have issues with the book.

Raising issues with the book

If you get stuck following any of the instructions in this book, or if you spot a mistake in the text or the code in the solutions, please raise an issue in the GitHub repository:

1. Use your favorite browser to navigate to the following link: <https://github.com/markjprice/apps-services-net10/issues>.
2. Click **New Issue**.
3. Enter as much detail as possible that will help me to diagnose the issue.

For example:

- The specific section title, page number, and step number.
- Your code editor, for example, Visual Studio, VS Code, or something else, including the version number.
- As much of your code and configuration that you feel is relevant and necessary.
- A description of the expected behavior and the behavior experienced.
- Screenshots (you can drag and drop image files into the issue box).

The following is less relevant but might be useful:

- Your operating system, for example, Windows 11 64-bit, or macOS Big Sur version 11.2.3.
- Your hardware, for example, Intel, Apple silicon, or ARM CPU.

I want all my readers to be successful with my book, so if I can help you (and others) without too much trouble, then I will gladly do so.

Giving me feedback

If you'd like to give me more general feedback about the book, then you can email me at markjprice@gmail.com. My publisher, Packt, has set up Discord channels for readers to interact with authors and other readers. You are

welcome to join us at the following link: https://packt.link/apps_and_services_dotnet10.

I love to hear from my readers about what they like about my books, as well as suggestions for improvements and how they are working with C# and .NET, so don't be shy. Please get in touch!

Thank you in advance for your thoughtful and constructive feedback.

Downloading solution code from the GitHub repository

I use GitHub to store solutions to all the hands-on, step-by-step coding examples throughout chapters and the practical exercises that are featured at the end of each chapter. You will find the repository at the following link: <https://github.com/markjprice/apps-services-net10>.

If you just want to download all the solution files without using Git, click the green **Code** button and then select **Download ZIP**.

I recommend that you add the preceding link to your favorites or bookmarks.

Good practice: It is best to clone or download the code solutions to a short folder path, like `C:\cs14net10\` or `C:\book\`, to avoid build-generated files exceeding the maximum path length. You should also avoid special characters like `#`. For example, do not use a folder name like `C:\C#projects\`. That folder name might work for a simple console app project, but once you start adding features that automatically generate code, you are likely to have strange issues. Keep your folder names short and simple.

Where to go for help

This section is about how to find quality information about programming on the web. You will learn about Microsoft Learn documentation, including its new MCP server for integration with AI systems, getting help while coding and using `dotnet` commands, getting help from fellow readers in the book's Discord channel, searching the .NET source code for implementation details, and finally, making the most of modern AI tools like GitHub Copilot.

This is useful information that all readers should know and refer to throughout reading any of my .NET 10 books, especially if you are new to .NET.

This section has been made into a separate *Appendix C*, both to make it reusable in all four of my .NET 10 books, and to avoid wasting pages in the print book for those readers who have already read it in one of the other books.

An online Markdown version of *Appendix C* is available in the book's GitHub repository at the following link: <https://github.com/markjprice/markjprice/blob/main/articles/getting-help.md>. Since this is hosted in my personal GitHub account, I can keep it updated more frequently throughout the three-year support period of .NET 10.

As part of this book's free exclusive benefits, you can also access a PDF version of the book, which includes *Appendix C*. You can unlock it and the other benefits at the following link: <https://packtpub.com/unlock>, then search for this book by name. Ensure it's the correct edition. Have your purchase invoice ready before you start.

My books are specifically written in a tutorial style so that you can get started quickly and confidently before then switching to the official documentation for more details or specialized scenarios.

This technique is commonly used by developers to learn how to code. More than 50% use books or other physical media, and then more than 80% use other online resources, as shown in the Stack Overflow Survey 2025 at the following link: <https://survey.stackoverflow.co/2025/>

[developers#2-how-did-you-learn-to-code.](#)

Subscribing to the official .NET blog and announcements

To keep up to date with .NET, an excellent blog to subscribe to is the official .NET blog, written by the .NET engineering teams, which you can find at the following link: <https://devblogs.microsoft.com/dotnet/>.

I also recommend that you subscribe to the official .NET announcements repository at the following link: <https://github.com/dotnet/announcements>.

Reading documentation on Microsoft Learn

The definitive resource for getting help with Microsoft developer tools and platforms is in the technical documentation on Microsoft Learn, which you can find at the following link: <https://learn.microsoft.com/en-us/docs>.

Getting help for the dotnet tool

At the command prompt, you can ask the `dotnet` tool for help with its commands. The syntax is:

```
dotnet help <command>
```

This will cause your web browser to open a page in the documentation about the specified command. Common `dotnet` commands include `new`, `build`, and `run`.

Warning! The `dotnet help new` command

worked with .NET Core 3.1 to .NET 6, but it returns an error with .NET 7 or later: Specified command 'new' is not a valid SDK command. Specify a valid SDK command. For more information, run `dotnet help`. Hopefully, the .NET team will fix that bug soon!

Another type of help is command-line documentation. It follows this syntax:

```
dotnet <command> -?|-h|--help
```

For example, `dotnet new -?` or `dotnet new -h` or `dotnet new --help` outputs documentation about the `new` command at the command prompt.

As you should now expect, `dotnet help` opens a web browser for the `help` command, and `dotnet help -h` outputs documentation for the `help` command at the command prompt!

Let's try some examples:

1. To open the official documentation in a web browser window for the `dotnet build` command, enter the following at the command prompt or in the VS Code terminal, and note the page opened in your web browser, as shown in *Figure 1.13*:

```
dotnet help build
```

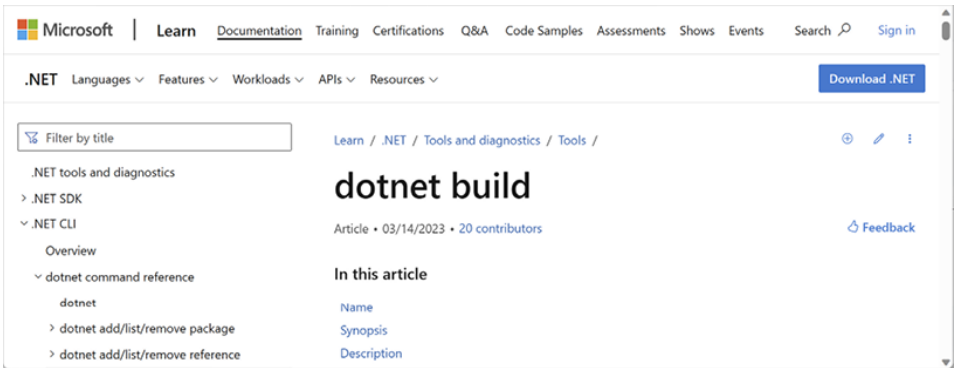


Figure 1.13: Web page documentation for the dotnet build command

1. To get help output at the command prompt, use the `-?`, `-h`, or `--help` flag, as shown in the following command:

```
dotnet build -?
```

2. You will see the following partial output:

```
Description: .NET Builder Usage: dotnet
build [<PROJECT | SOLUTION>...] [options]
Arguments: <PROJECT | SOLUTION> The
project or solution file to operate on. If
a file is not specified, the command will
search the current directory for one.
Options: --ucr, --use-current-runtime Use
current runtime as the target runtime. -f,
--framework <FRAMEWORK> The target
framework to build for. The target
framework must also be specified in the
project file. ... -?, -h, --help Show
command-line help.
```

3. Repeat both types of help request for the following commands: `add`,

help, list, new, and run.

Getting help on Discord and other chat forums

Asking questions in programming forums and Discord channels is an art as much as it is a science. To maximize your chances of receiving a helpful answer, there's a blend of clarity, specificity, and community awareness that you should aim for.

Here are some tips for asking questions:

- **Ask in a public channel, not in private. Please do not direct message an author with a question or a friend request.** Remember, every question asked and answered builds the collective knowledge and resourcefulness of the whole community. Asking in public also allows other readers to help you, not just the author. The community that Packt and I have built around my books is friendly and smart. Let us *all* help you.
- **Research before asking:** It's important to look for answers yourself before turning to the community. Use search engines, official documentation, and the search function within the forum or Discord server. This not only respects the community's time but also helps you learn more effectively. Another place to look first is the errata and improvements section of the book, found at the following link: <https://github.com/markjprice/cs14net10/blob/main/docs/errata/README.md>.
- **Be specific and concise:** Clearly state what you're trying to achieve, what you've tried so far, and where you're stuck. A concise question is more likely to get a quick response.
- **Specify the book location:** If you are stuck on a particular part of the book, specify the page number and section title so that others can look up the context of your question.

- **Show your work:** Demonstrating that you've made an effort to solve the problem yourself not only provides context but also helps others understand your thought process and where you might have gone down a wrong path.
- **Prepare your question:** Avoid too broad or vague questions. Screenshots of errors or code snippets (with proper formatting) can be very helpful.

Oddly, I've been seeing more and more examples of readers taking photos of their screens and posting those. These are harder to read and limited in what they can show. It's better to copy and paste the text of your code or the error message so that others can copy and paste it themselves. Alternatively, at least take a high-resolution screenshot instead of a photo with your phone camera at a jaunty angle!

- **Format your code properly:** Most forums and Discord servers support code formatting using Markdown syntax. Use formatting to make your code more readable. For example, surround code keywords in single backticks, like ``public void``, and surround code blocks with three backticks with optional language code, as shown in the following code:

```
```cs using static System.Console;  
WriteLine("This is C# formatted code.");
```
```

Good practice: After the three backticks that start a code block in Markdown,

specify a language short name like `cs`,
`csharp`, `js`, `javascript`, `json`,
`html`, `css`, `cpp`, `xml`, `mermaid`,
`python`, `java`, `ruby`, `go`, `sql`,
`bash`, or `shell`.

To learn how to format text in Discord
channel messages, see the following link:
[https://support.discord.com/
hc/en-us/articles/210298617-
Markdown-Text-101-Chat-
Formatting-Bold-Italic-
Underline](https://support.discord.com/hc/en-us/articles/210298617-Markdown-Text-101-Chat-Formatting-Bold-Italic-Underline).

- **Be polite and patient:** Remember, you're asking for help from people who are giving their time voluntarily. A polite tone and patience while waiting for a response go a long way. Channel participants are often in a different time zone, so you may not see your question answered until the next day.
- **Be ready to actively participate:** After asking your question, stay engaged. You might receive follow-up questions for clarification. Responding promptly and clearly can significantly increase your chances of getting a helpful answer. When I ask a question, I set an alarm for three hours later to go back and see if anyone has responded. If there hasn't been a response yet, then I set another alarm for 24 hours later.

Incorporating these approaches when asking questions not only increases your likelihood of getting a useful response but also contributes positively to the community, by showing respect for others' time and effort.

Good practice: Never just say "Hello" as a message on any chat system. You can read why at the following link: <https://nohello.net/>.

Similarly, don't ask to ask: <https://dontasktoask.com/>.

AI tools like ChatGPT and GitHub Copilot

One of the biggest changes in coding and development in the past year is the emergence of generative **artificial intelligence (AI)** tools that can help with coding tasks like completing a code statement, implementing an entire function, writing unit tests, and suggesting debugging fixes for existing code.

You can read what developers say about AI tools in the 2025 Stack Overflow Developer Survey: “84% of respondents are using or plan to use AI tools in their development process” (<https://survey.stackoverflow.co/2025/ai/#1-ai-tools-in-the-development-process/>).

ChatGPT

Let's say you need to understand more about how to design a microservice. You might enter the following prompt:

```
You are an expert .NET solution architect with
experience advising global enterprises. What
are the top three most common mistakes when
designing a microservice architecture that are
serious enough to cause a project to be
abandoned?
```

It responds with a summary of those top three mistakes, as shown in detail in *Figure 1.14* and the following list:

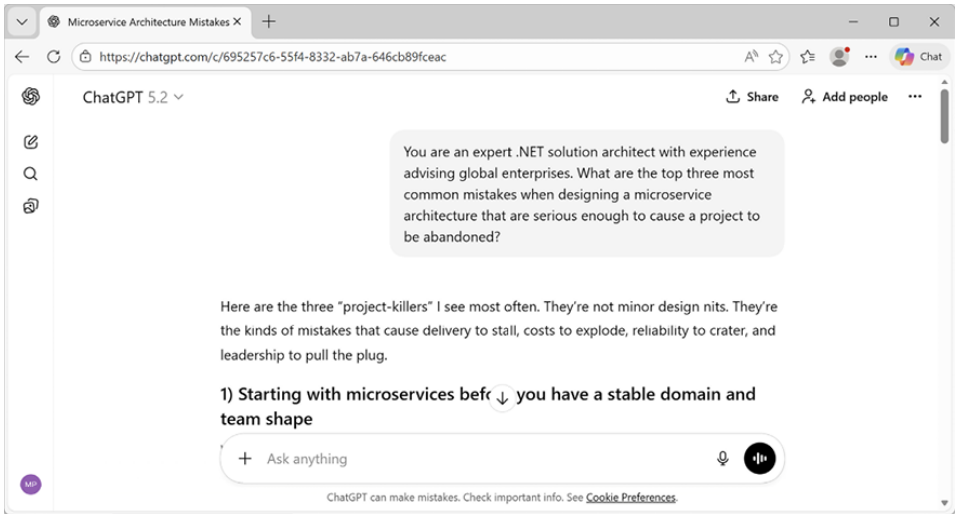


Figure 1.14: ChatGPT advising about microservice architecture design

"Here are the three "project-killers" I see most often. They're not minor design nits. They're the kinds of mistakes that cause delivery to stall, costs to explode, reliability to crater, and leadership to pull the plug.

1. Starting with microservices before you have a stable domain and team shape

- The most common failure is deciding "we're doing microservices" before the business domain is properly understood and before teams are set up to own services end to end. Services get split along technical lines or org charts instead of real business capabilities. The result is constant cross-service changes, endless coordination, and slower delivery than the monolith ever had.
- This kills projects because microservices only work when teams can move independently.

If every change touches five services owned by three teams, the architecture becomes the problem. Leadership eventually sees rising costs and falling velocity and pulls the plug.

4. Designing services but keeping monolithic data habits

- Another fatal mistake is letting services share databases, query each other's tables, or rely on distributed transactions to keep everything perfectly consistent. This preserves tight coupling while adding all the complexity of a distributed system. Deployments stop being independent, schema changes become dangerous, and data bugs turn into business incidents.
- Projects die here because the system becomes fragile and impossible to reason about. Teams spend their time firefighting data issues and performance problems instead of delivering features, and confidence in the architecture collapses.

7. Ignoring the operational cost of microservices

- Many teams build lots of services without building the platform needed to run them. There's weak CI/CD, poor logging, no distributed tracing, and no clear way to diagnose failures across services. Production issues take hours or days to understand, and releases become stressful events.

- This sinks projects because microservices move complexity from code to operations. If observability, automation, and reliability aren't in place early, the system becomes unmanageable. Eventually the organization decides the complexity isn't worth it and abandons the approach."

As well as high-level design, LLMs can help you write code and tests, as well as solve problems using debugging tools.

GitHub Copilot for programmers

Microsoft has a service specifically for programmers named GitHub Copilot that can help autocomplete code directly in your code editor. It has plugins for code editors including Visual Studio, VS Code, and JetBrains IntelliJ-based IDEs.

Personally, I really like the Copilot branding. It makes it clear that you are the pilot. You are ultimately responsible for “flying the plane.” But for the easy or boring bits, you can hand it over to your co-pilot for a bit, while being actively ready to take back control if needed.

For individuals, **GitHub Copilot** has a free plan, and then a paid Pro plan at \$10 per month or \$100 per year, and a Pro+ plan at \$39 per month or \$390 per year. You can find the latest pricing information at the following link: <https://github.com/features/copilot/plans>.

You should check online for which Copilot features are available for various code editors. As you can imagine, this is a fast-changing world and a lot of what I might write in the book today will be out of date by the time you read it: <https://github.com/features/copilot>.

In June 2025, Visual Studio switched to GPT-4.1 as its default model and added

support for many more models to choose from:

- Claude Sonnet 4.5, Claude Opus 4.5, Claude Haiku 4.5
- GPT-5.2-Codex and other GPT-5.x family models for Copilot Pro subscribers
- Gemini 3 Flash, Gemini 3 Pro

They each have strengths, and you can learn how to choose the right model for your task at the following link:

<https://docs.github.com/en/copilot/reference/ai-models/model-comparison>.

For premium requests, you have a monthly quota. Use the **Copilot Consumptions** panel to help you monitor your usage. If you run out of premium requests, it will automatically switch to GPT-4.1 at no extra cost.

JetBrains has its own equivalent named AI Assistant, which you can read about at the following link: <https://blog.jetbrains.com/idea/2023/06/ai-assistant-in-jetbrains-ides/>.

You can sign up for GitHub Copilot at the following link:

<https://github.com/github-copilot/signup/>

Good practice: Learn more about how to use Copilot as your coding GPS at the following link: <https://devblogs.microsoft.com/visualstudio/using-github-copilot-as-your-coding-gps/>.

Customizing Copilot responses

To get the best out of AI you must supply it with clear, detailed instructions. A

prompt like, create a .NET project for a social media website, gives the LLM plenty of scope for hallucinations. But giving all the necessary details in every prompt is a pain. What we need is a way to give standard instructions that apply to all projects, so then we only need to give specific instructions in the actual prompt.

We can do this for Copilot using a special Markdown file:

```
<project_folder>\.github\copilot-  
instructions.md
```

Custom instructions define:

- How tasks should be performed.
- Which technologies are preferred.
- What project structure should be followed.
- What to check for in a code review.

For example:

```
--- description: 'Guidelines for building C#  
applications' applyTo: '**/*.cs' --- # C#  
Development ## C# Instructions - Always use  
the latest version C#, currently C# 14  
features. - Write clear and concise comments  
for each function. ## General Instructions -  
Make only high confidence suggestions when  
reviewing code changes. - Write code with good  
maintainability practices, including comments  
on why certain design decisions were made. -  
Handle edge cases and write clear exception  
handling. - For libraries or external  
dependencies, mention their usage and purpose  
in comments. ## Naming Conventions - Follow
```

PascalCase for component names, method names, and public members. - Use camelCase for private fields and local variables. - Prefix interface names with "I" (e.g., IUserService).

The full file is available at the following link:

<https://raw.githubusercontent.com/github/awesome-copilot/refs/heads/main/instructions/csharp.instructions.md>.

These instructions are automatically incorporated with every chat request. You will see the file referenced in the AI chat window during a request.

Here are some more links about using AI for coding:

- OpenAI's guide, Text generation and prompting: <https://platform.openai.com/docs/guides/prompt-engineering>
- A Beginner's Guide to Prompt Engineering with GitHub Copilot: <https://dev.to/github/a-beginners-guide-to-prompt-engineering-with-github-copilot-3ibp>
- Awesome GitHub Copilot Customizations, A curated collection of prompts, instructions, and chat modes to supercharge your GitHub Copilot experience across different domains, languages, and use cases: <https://github.com/github/awesome-copilot/>
- The Register article about AI: https://www.theregister.com/2024/01/27/ai_coding_automatic/
- Getting Started with GitHub Copilot in Visual Studio (YouTube playlist): <https://www.youtube.com/playlist?list=PLReL099Y5nRfCULHLrLAgly0JF6F5L6S1>
- StackOverflow 2025 survey – AI in the development workflow:

<https://survey.stackoverflow.co/2025/ai/#2-ai-in-the-development-workflow>

Disabling tools when they get in the way

Although these tools can be helpful, they can also get in your way, especially when learning, because they sometimes do work for you without telling you. If you do not do that work for yourself at least a few times, you won't learn fully.

To configure IntelliSense for C# in Visual Studio:

1. Navigate to **Tools | Options**.
2. In the **Options** dialog box tree view, navigate to **Text Editor | C# | IntelliSense**.
3. Click the ? button in the caption bar to view the documentation.

To configure GitHub Copilot in Visual Studio:

1. Navigate to **Tools | Options**.
2. In the **Options** dialog box tree view, navigate to **GitHub | Copilot**.
3. Set **Enable Globally** to **True** or **False**, and then click **OK**.

To disable GitHub Copilot in VS Code:

1. In the status bar, on the right, to the left of the notification icon, click the GitHub Copilot icon.
2. In the popup, click **Disable Globally**.
3. To enable it, click the GitHub Copilot icon again and then click **Enable Globally**.

Using future versions of .NET with this book

Microsoft is expected to release .NET 11 at the .NET Conf 2026 on Tuesday, November 10, 2026. Many readers will want to use this book with .NET 11 and future versions of .NET, so this section explains how. Since .NET 11 will be an

STS release with two years of support, it will reach its end of life on the same day as .NET 10 in November 2028. This means organizations can upgrade to .NET 11 and gain its performance improvements even if they don't use any new features.

.NET 11 is likely to be available in preview from February 2026, or you can wait for the final version in November 2026.

Warning! Once you install a .NET 11 SDK, then it will be used by default for all .NET projects unless you override it using a `global.json` file. You can learn more about doing this at the following link: <https://learn.microsoft.com/en-us/dotnet/core/tools/global-json>.

You can easily continue to target the .NET 10 runtime while installing and using future C# compilers, as shown in *Figure 1.15* and illustrated in the following list:

1. **November 2025 onwards:** Install .NET SDK 10.0.100 or later and use it to build projects that target .NET 10 and use the C# 14 compiler by default. Every month, update to .NET 10 SDK patches on the development computer and update to .NET 10 runtime patches on any deployment computers.
2. **February to October 2026:** Optionally, install .NET SDK 11 previews each month to explore the new C# 15 language and .NET 11 library features. Note that you won't be able to use new library features while targeting .NET 10.
3. **November 2026 onwards:** Install .NET SDK 11.0.100 or later and use it to build projects that continue to target .NET 10 and use the C# 15 compiler for its new features. You will be using a fully supported SDK and fully supported runtime. You can also use new features in EF Core

11 because it will continue to target .NET 10.

4. **February to October 2027:** Optionally, install .NET 12 previews to explore new C# 16 language and .NET 12 library features. Start planning if any new library and ASP.NET Core features in .NET 11 and .NET 12 can be applied to your .NET 10 projects when you are ready to migrate.
5. **November 2027 onwards:** Install .NET 12.0.100 SDK or later and use it to build projects that target .NET 10 and use the C# 16 compiler.
6. You could migrate your .NET 10 projects to .NET 12 since .NET 12 is a **long-term support (LTS)** release. You have until November 2028 to complete the migration when .NET 10 reaches end-of-life.

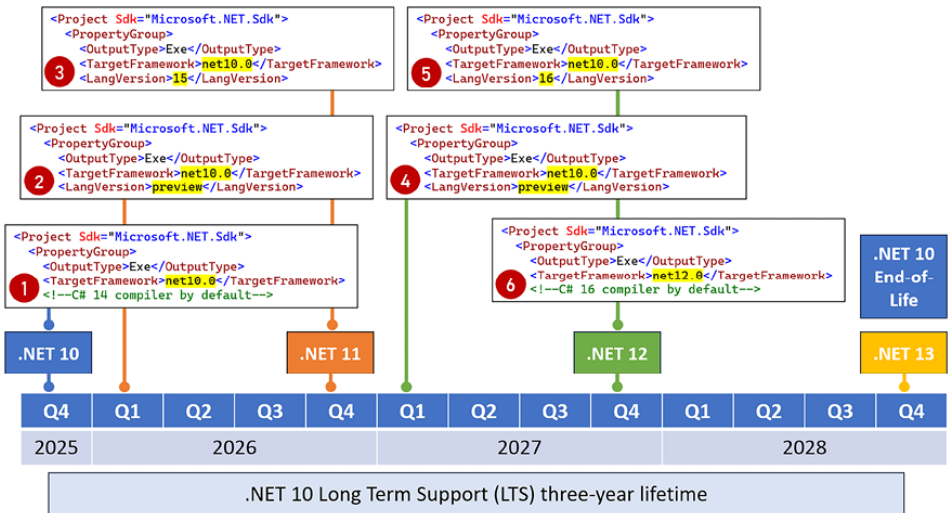


Figure 1.15: Targeting .NET 10 for long-term support while using the latest C# compilers

When deciding to install a .NET SDK, remember that the latest is used by default to build any .NET projects. Once you've installed a .NET 11 SDK preview, it will be used by default for all projects, unless you force the use of an older, fully supported SDK version like 10.0.100 or a later patch.

To gain the benefits of whatever new features are available in C# 15, while still targeting .NET 10 for long-term support, modify your project file, as shown highlighted in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk">
<PropertyGroup> <OutputType>Exe</OutputType>
<TargetFramework>net10.0</TargetFramework>
<LangVersion>15</LangVersion> <!--Requires
.NET 11 SDK GA--> <ImplicitUsings>enable</
ImplicitUsings> <Nullable>enable</Nullable> </
PropertyGroup> </Project>
```

Good practice: Use a **general availability (GA)** SDK release like .NET 11 to use new compiler features while still targeting older but longer supported versions of .NET like .NET 10.

Practicing and exploring

Test your knowledge and understanding by answering some questions, getting some hands-on practice, and exploring with deeper research the topics in this chapter.

Exercise 1.1 – Test your knowledge

Use the web to answer the following questions:

1. Why is it good practice to add the following setting to your project files?
And when should you not set it?

```
<TreatWarningsAsErrors>true</
TreatWarningsAsErrors>
```

2. Which service technology requires a minimum HTTP version of 2?
3. In 2010, your organization created a service using .NET Framework and Windows Communication Foundation. What is the best technology to

migrate to and why?

4. Which code editor or IDE should you install for .NET development?
5. What should you beware of when creating Azure resources?

Appendix A, Answers to the Test Your Knowledge Questions, is available to download from the following link: https://github.com/markjprice/apps-services-net10/blob/main/docs/B31467_Appendix%20A.pdf.

Exercise 1.2 – Explore topics

Use the links on the following webpage to learn more about the topics covered in this chapter: <https://github.com/markjprice/apps-services-net10/blob/main/docs/book-links.md#chapter-1---introducing-apps-and-services-with-net>.

Summary

In this chapter, you:

- Were introduced to the app and service technologies that you will learn about in this book.
- Set up your development environment.
- Learned where to look for help.

In the next chapter, you will learn how to build mobile apps using .NET MAUI.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow this QR code:

https://packt.link/apps_and_services_dotnet10



Join .NETPro – It's Free

Staying sharp in .NET takes more than reading release notes. It requires real-world tips, proven patterns, and scalable solutions. That's what .NETPro, Packt's new newsletter, is all about.

Scan the QR code or visit the link to subscribe:

<https://landing.packtpub.com/dotnetpronewsletter/>



2

Building Mobile Apps Using .NET MAUI

Cross-platform GUI development using .NET MAUI, Avalonia, and Blazor cannot be learned in only a hundred or so pages, but I want to introduce you to some of what is possible. Think of the three cross-platform app chapters and the additional online-only sections as an introduction that will give you a taste to inspire you; then you can learn more from a book dedicated to web, mobile, or desktop app development.

This chapter is about learning how to make **graphical user interface (GUI)** apps by building a cross-platform mobile app for iOS and Android using **.NET MAUI (Multi-platform App UI)**. Although you can create desktop apps for macOS Catalyst and Windows using .NET MAUI, Avalonia is better for that purpose, so we will cover desktop apps in *Chapter 3, Building Desktop Apps Using Avalonia*.

In this chapter, you will first see how **eXtensible Application Markup Language (XAML)** makes it easy to define the **user interface (UI)** for a graphical app. XAML is pronounced “zamel.” Then we will review .NET MAUI and how it helps to build cross-platform mobile apps, including how to share resources and perform data binding.

The mobile app that you will build allows the listing and management of customers in the Northwind database.

A Windows computer with Visual Studio version 17.14 or later, or any operating system with VS Code and the `dotnet` CLI or Rider, can be used to create a .NET MAUI project. But you will need a computer with macOS and Xcode to compile for iOS.

In this chapter, we will cover the following topics:

- Understanding XAML
- Understanding .NET MAUI
- Building mobile apps using .NET MAUI
- Using shared resources
- Introducing data binding

Understanding XAML

Let's start by looking at XAML, the markup language used by .NET MAUI.

In 2006, Microsoft released **Windows Presentation Foundation (WPF)**, which was the first technology to use XAML. Silverlight, for web and mobile apps, quickly followed, but it is no longer supported by Microsoft. WPF is still used today to create Windows desktop applications with .NET 10.

XAML can be used to build parts of the following apps:

- **.NET MAUI apps** for mobile and desktop devices, including Android, iOS, Windows, and macOS. It is an evolution of a technology named **Xamarin.Forms**.
- **WinUI 3 apps** for Windows 10 and 11.
- **Universal Windows Platform (UWP) apps** for Windows 10 and 11, Xbox, Mixed Reality, and Meta Quest VR headsets.

- **WPF apps** for Windows desktop, including Windows 7 and later.
- **Avalonia** and **Uno Platform apps** using cross-platform third-party technologies.

Simplifying code using XAML

XAML simplifies C# code, especially when building a UI.

Imagine that you need two or more pink buttons laid out horizontally to create a toolbar, which execute a method for their implementation when clicked.

In C#, you might write the following code:

```
HorizontalStackPanel toolbar = new(); Button
newButton = new(); newButton.Content = "New";
newButton.Background = new
SolidColorBrush(Colors.Pink);
newButton.Clicked += NewButton_Clicked;
toolbar.Children.Add(newButton); Button
openButton = new(); openButton.Content =
"Open"; openButton.Background = new
SolidColorBrush(Colors.Pink);
openButton.Clicked += OpenButton_Clicked;
toolbar.Children.Add(openButton);
```

In XAML, this could be simplified to the following lines of code:

```
<HorizontalStackPanel x:Name="toolbar">
<Button x:Name="newButton" Background="Pink"
Clicked="NewButton_Clicked">New</Button>
<Button x:Name="openButton" Background="Pink"
Clicked="OpenButton_Clicked">Open</Button> </
StackPanel>
```

When this XAML is processed, the equivalent properties are set, and methods are called to achieve the same goal as the preceding C# code.

You can think of XAML as an alternative and easier way of declaring and instantiating .NET types, especially when defining a UI and the resources that it uses.

XAML allows resources like brushes, styles, and themes to be declared at different levels, like a UI element or a page, or globally for the application to enable resource sharing.

XAML allows data binding between UI elements or between UI elements and objects and collections.

If you choose to use XAML to define your UI and related resources at compile time, then the code-behind file must call the `InitializeComponent` method in the page constructor, as shown highlighted in the following code:

```
public partial class MainPage : ContentPage {
    public MainPage() { InitializeComponent(); //
        Process the XAML markup. } private void
    NewButton_Clicked(object sender, EventArgs e)
    { ... } private void OpenButton_Clicked(object
    sender, EventArgs e) { ... } }
```

Calling the `InitializeComponent` method tells the page to read its XAML, create the controls defined in it, and set their properties and event handlers.

.NET MAUI namespaces

.NET MAUI has several important namespaces where its types are defined, as shown in *Table 2.1*:

Name		Description	

local, so when an instance of the `CustomerList` control is needed, it is declared using `<local:CustomerList>`.

You can import as many namespaces with different prefixes as you need.

Type converters

Type converters convert XAML attribute values that must be set as `string` values into other types. For example, the following button has its `Background` property set to the `string` value `"Pink"`:

```
<Button x:Name="newButton" Background="Pink"
...

```

This is converted into a `SolidColorBrush` instance using a type converter, as shown in the following equivalent code:

```
newButton.Background = new
SolidColorBrush(Colors.Pink);
```

There are many type converters provided by .NET MAUI and you can create and register your own. These are especially useful for custom data visualizations.

Markup extensions

To support some advanced features, XAML uses markup extensions. Some of the most important enable element and data binding and the reuse of resources, as shown in the following list:

- `{Binding}` links an element to a value from another element or a data source.
- `{OnPlatform}` sets properties to different values depending on the

current platform.

- {StaticResource} and {DynamicResource} link an element to a shared resource.
- {AppThemeBinding} links an element to a shared resource defined in a theme.

.NET MAUI provides the `OnPlatform` markup extension to allow you to set different markup depending on the platform. For example, the iPhone X introduced the notch that takes up extra space at the top of the phone display. We could add extra padding to an app that applies to all devices, but it would be better if we could add that extra padding only to iOS, as shown in the following markup:

```
<VerticalStackLayout>
<VerticalStackLayout.Padding> <OnPlatform
x:TypeArguments="Thickness"> <On
Platform="iOS" Value="30,60,30,30" /> <On
Platform="Android" Value="30" /> <On
Platform="WinUI" Value="30" /> </OnPlatform>
</VerticalStackLayout.Padding>
```

There is a simplified syntax too, as shown in the following markup:

```
<VerticalStackLayout Padding="{OnPlatform
iOS='30,60,30,30', Default='30'}">
```

Choosing between .NET MAUI controls

There are lots of predefined controls that you can choose from for common UI scenarios. .NET MAUI (and most dialects of XAML) support these controls, as shown in *Table 2.2*:

--	--	--

Description		
Executing actions	Button, MenuItem, ToolbarItem	
Choosing options	RadioButton, Switch	
Choosing dates and times	Picker	
Choosing items from lists and tables	Picker, TableView	
Scrolling	ScrollView	
Showing animated views that show one item at a time	CarouselView	
Layout containers that affect their child controls in different ways	Grid, HorizontalStackLayout, StackLayout, VerticalStackLayout	
Visual elements	ContentView, Frame, ScrollView	
Graphical elements	Path, Polygon, Polyline, Rectangle, RoundedRectangle	
Displaying read-only text and other read-only displays	RefreshView	
Editing text	Entry	
Embedding images, videos, and audio files		
Selecting with pages of numbers		
Adding a search feature		
Embedding Blazor and web components		
Building custom controls		

Table 2.2: MAUI user interface controls

- .NET MAUI defines its controls in the `Microsoft.Maui.Controls` namespace. It has some specialized controls too:
- **Application:** Represents a cross-platform graphical application. It sets the root page, manages windows, themes, and resources, and provides app-level events like `PageAppearing`, `ModalPushing`, and `RequestedThemeChanged`. It also has methods that you can override to hook into app events like `OnStart`, `OnSleep`, `OnResume`, and `CleanUp`.
 - **Shell:** A `Page` control that provides UI features that most applications require, like flyout or tab bar navigation, navigation tracking and

management, and navigation events.

Most .NET MAUI controls derive from `View`. One of the most important characteristics of a `View`-derived type is that they can be nested. This allows you to build complex custom user interfaces.

Understanding .NET MAUI

To create a mobile app that only needs to run on iPhones, you might choose to build it with either the Objective-C or Swift language and the UIKit libraries using the Xcode development tool.

To create a mobile app that only needs to run on Android phones, you might choose to build it with either the Java or Kotlin language and the Android SDK libraries using the Android Studio development tool.

But what if you need to create a mobile app that can run on iPhones *and* Android phones? And what if you only want to create that mobile app once using a programming language and development platform that you are already familiar with? And what if you realized that with a bit more coding effort to adapt the UI to desktop-size devices, you could target macOS and Windows desktops too?

.NET MAUI enables developers to build cross-platform mobile apps for Apple iOS (iPhone), iPadOS, macOS using Catalyst, Windows using WinUI 3, and Google Android using C# and .NET, which are then compiled to native APIs and executed on native phone and desktop platforms.

Business logic layer code can be written once and shared between all platforms. UI interactions and APIs are different on various mobile and desktop platforms, so the UI layer is sometimes customized for each platform.

Like WPF and UWP apps, .NET MAUI uses XAML to define the UI once for all platforms using abstractions of platform-specific UI components.

Applications built with .NET MAUI draw the UI using native platform widgets, so the app's look and feel fits naturally with the target mobile platform.

A user experience built using .NET MAUI will not perfectly fit a specific platform in the same way that one custom built with native tools for that platform would, but for mobile and desktop apps that will not have millions of users, it is good enough. With some effort, you can build beautiful apps, as illustrated by the Microsoft challenge that you can read about at the following link: <https://devblogs.microsoft.com/dotnet/announcing-dotnet-maui-beautiful-ui-challenge/>.

.NET MAUI and Xamarin support

Major versions of .NET MAUI ship with .NET starting with .NET 7 but as an optional workload. This means that .NET MAUI does not follow the same **Short Term Support (STS)/Long Term Support (LTS)** as the main .NET platform. Every version of .NET MAUI only has 18 months of support, so .NET MAUI effectively is always an STS release, and this includes the .NET MAUI version that ships with .NET 10.

.NET MAUI has dependencies on other OSes like iOS and macOS so it gets complicated. Major versions of iOS usually release in September, and major versions of iPadOS and macOS often release later in October or November. This does not give the .NET MAUI team much time to make sure their platform works well with those operating systems before a major version of .NET is released in early November.

Warning! Xamarin reaches its **end-of-life (EOL)** on May 1, 2024 so any Xamarin and Xamarin.Forms projects should migrate to .NET MAUI or an alternative like Avalonia or Uno.

Development tools for mobile first, cloud

first

Mobile apps are often supported by services in the cloud.

Satya Nadella, CEO of Microsoft, famously said the following:

To me, when we say mobile first, it's not the mobility of the device, it's actually the mobility of the individual experience. [...] The only way you are going to be able to orchestrate the mobility of these applications and data is through the cloud.

When installing Visual Studio, you must select the **.NET Multi-platform App UI development** workload, which is in the **Desktop & Mobile** section, as shown in *Figure 2.1*:

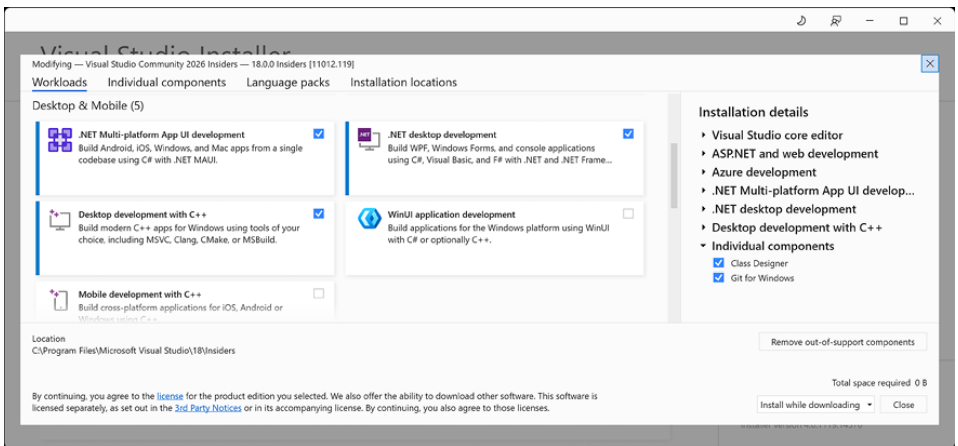


Figure 2.1: Selecting the .NET MAUI workload for Visual Studio

Installing .NET MAUI workloads manually

Installing Visual Studio should install the required .NET MAUI workloads if you selected them. If not, then you can make sure that the workloads are installed manually.

To see which workloads are currently installed, enter the following command:

```
dotnet workload list
```

The currently installed workloads will appear in a table, as shown in the following output:

```
Workload version: 10.0.100-manifests.a6e8bec0
Installed Workload Id Manifest Version
Installation Source
-----
android 36.0.0/10.0.100 VS 18.0.11012.119 ios
18.5.10727-net10/10.0.100 VS 18.0.11012.119
maccatalyst 18.5.10727-net10/10.0.100 VS
18.0.11012.119 maui-windows 10.0.0/10.0.100 VS
18.0.11012.119 wasm-tools 10.0.100/10.0.100 VS
18.0.11012.119
```

To install the .NET MAUI workloads for all platforms, enter the following command at the command line or terminal:

```
dotnet workload install maui
```

To update all existing workload installations, enter the following command:

```
dotnet workload update
```

To add missing workload installations required for a project, in the folder containing the project file, enter the following command:

```
dotnet workload restore <projectname>
```

Over time you are likely to install multiple versions of workloads related to different versions of .NET SDKs. Before .NET 8, developers tried to manually delete workload folders, which can cause problems. Introduced with .NET 8 was a new feature to remove leftover and unneeded workloads, as shown in the following command:

```
dotnet workload clean
```

If you want to use Visual Studio to create an iOS mobile app or a macOS Catalyst desktop app, then you can connect over a network to a **Mac build host**. Instructions can be found at the following link: <https://learn.microsoft.com/en-us/dotnet/maui/ios/pair-to-mac>.

.NET MAUI user interface component categories

.NET MAUI includes some common controls for building user interfaces. These can be divided into four categories:

- **Pages** represent cross-platform application screens, for example, `Shell`, `ContentPage`, `NavigationPage`, `FlyoutPage`, and `TabbedPage`.
- **Layouts** represent the structure of a combination of other UI components, for example, `Grid`, `StackLayout`, and `FlexLayout`.
- **Views** represent a single user interface component, for example, `CarouselView`, `CollectionView`, `Label`, `Entry`, `Editor`, and `Button`.
- **Cells** represent a single item in a list or table view, for example, `TextCell`, `ImageCell`, `SwitchCell`, and `EntryCell`.

Shell control

The `Shell` control is designed to simplify app development by providing standardized navigation and search capabilities. In your project, you would create a class that inherits from the `Shell` control class. Your derived class defines components like a `TabBar`, which contains `Tab` items, `FlyoutItem` instances, and `ShellContent`, which contain the `ContentPage` instances for each page. A `TabBar` should be used when there are only up to about four or five pages to navigate between. `FlyoutItem` navigation should be used when there are more items because they can be presented as a vertical scrollable list. You can use both, with the `TabBar` showing a subset of items. The `Shell` will keep them synchronized. Flyout navigation is when a list of items flies out (or slides) from the left side of a mobile device's screen or desktop app's main window. The user invokes it by tapping on a "hamburger" icon with three horizontal lines stacked on top of each other. When the user taps a flyout item, its page is instantiated when needed, as the user navigates around the UI.

The top bar automatically shows a **Back** button when needed to allow the user to navigate back to a previous page.

ListView control

The `ListView` control is used for long lists of data-bound values of the same type. It can have headers and footers, and its list items can be grouped. It has cells to contain each list item. There are two built-in cell types: text and image. Developers can define custom cell types. Cells can have context actions that appear when the cell is swiped on an iPhone, long-pressed on Android, or right-clicked on a desktop OS. A context action that is destructive can be shown in red, as shown in the following markup:

```
<TextCell Text="{Binding CompanyName}"
Detail="{Binding Location}">
```

```
<TextCell.ContextActions> <MenuItem  
Clicked="Customer_Phoned" Text="Phone" />  
<MenuItem Clicked="Customer_Deleted"  
Text="Delete" IsDestructive="True" /> </  
TextCell.ContextActions> </TextCell>
```

Entry and Editor controls

The `Entry` and `Editor` controls are used for editing text values and are often data-bound to an entity model property, as shown in the following markup:

```
<Editor Text="{Binding CompanyName,  
Mode=TwoWay}" />
```

Good practice: Use `Entry` for a single line of text. Use `Editor` for multiple lines of text.

.NET MAUI handlers

In .NET MAUI, XAML controls are defined in the `Microsoft.Maui.Controls` namespace. Components called **handlers** map these common controls to native controls on each platform. On iOS, a handler will map a .NET MAUI `Button` to an iOS-native `UIButton` defined by UIKit. On macOS, `Button` is mapped to `NSButton` defined by AppKit. On Android, `Button` is mapped to an Android-native `AppCompatActivity`.

Handlers have a `NativeView` property that exposes the underlying native control. This allows you to work with platform-specific features like properties, methods, and events, and customize all instances of a native control.

Writing platform-specific code

If you need to write code statements that only execute for a specific platform like Android, then you can use compiler directives.

For example, by default, `Entry` controls on Android show an underline character. If you want to hide the underline, you could write some Android-specific code to get the handler for the `Entry` control, use its `NativeView` property to access the underlying native control, and then set the property that controls that feature to `false`, as shown in the following code:

```
#if __ANDROID__
Handlers.EntryHandler.EntryMapper[nameof(IEntry.BackgroundColor)]
= (h, v) => { (h.NativeView as
global::Android.Views.Entry).UnderlineVisible
= false; }; #endif
```

Predefined compiler constants include the following:

- `__ANDROID__`
- `__IOS__`
- `WINDOWS`

The compiler `#if` statement syntax is slightly different from the C# `if` statement syntax, as shown in the following code:

```
#if __IOS__ // iOS-specific statements #elif
__ANDROID__ // Android-specific statements
#elif WINDOWS // Windows-specific statements
#endif
```

Now that you have been introduced to some of the important concepts around MAUI apps, and you've set up the additional components needed for MAUI, let's get practical and build a MAUI project.

Building mobile and desktop apps using .NET MAUI

We will build a mobile and desktop app for managing customers in Northwind. Although this chapter focuses on developing a mobile app, it can be useful to quickly try it out on Windows, especially since .NET MAUI gives us that capability “for free.”

Good practice: If you have a Mac and you have never run Xcode on it, then run it now until you see the *Start* window. This will ensure that all its required components are installed and registered. If you do not do this, then you might get errors with your projects later.

Creating a virtual Android device for local app testing

To target Android with a mobile app, you must install at least one Android SDK. A default installation of Visual Studio with the mobile development workload already includes one Android SDK, but it is often an older version designed to support as many Android devices as possible. Google recommends updating your Android SDK to the latest version to avoid common issues with the Android emulator.

To use the latest features of .NET MAUI, you must configure a more recent Android virtual device:

1. In Windows, start **Visual Studio**. If you see the **Visual Studio 2026** modal dialog box, then click **Continue without code**.
2. Navigate to **Tools | Android | Android Device Manager**. If you are prompted by **User Account Control** to allow this app to make changes to your device, click **Yes**.

3. In the **Android Device Manager**, click the **+ New** button to create a new device.
4. In the dialog box, make the following choices, as shown in *Figure 2.2*:
- **Base Device: Pixel 9 (+ Store)**
 - **Processor: x86_64**
 - **OS: API 36**
 - **Google APIs: Selected**
 - **Google Play Store: Cleared**

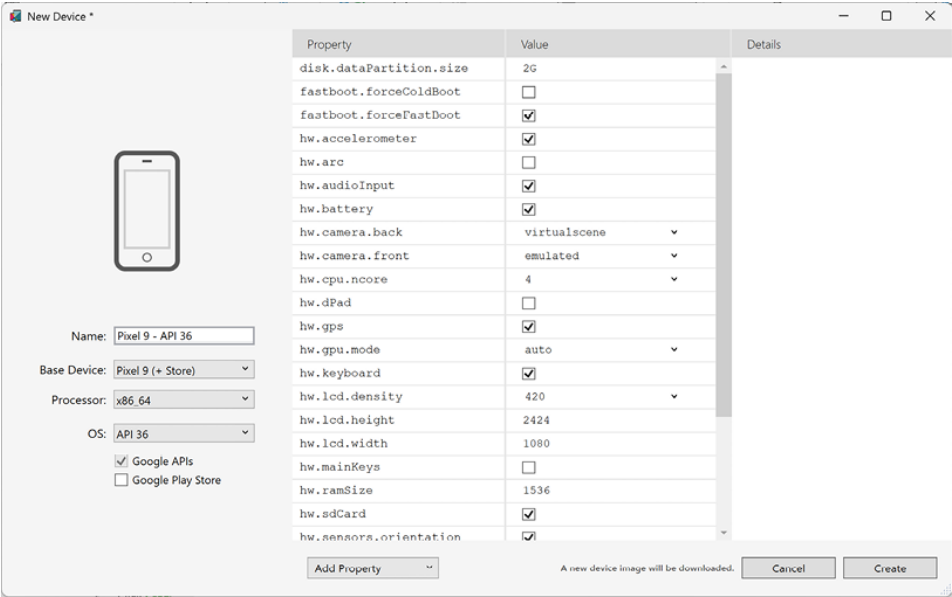


Figure 2.2: Selecting the hardware and OS for a virtual Android device

1. Click **Create**.
2. Accept any license agreements.
3. Wait for any required downloads.
4. In **Android Device Manager**, in the list of devices, in the row for the device that you just created, click **Start**.
5. Be patient! It can take a few minutes for the emulator to start.
6. When the Android device has finished starting, click the Chrome browser

and test that it has access to the network by navigating to <https://www.bbc.co.uk/news>.

7. Close the emulator.
8. Close **Android Device Manager**.
9. Restart Visual Studio to ensure that it is aware of the new emulator.

Enabling Windows developer mode

If you were only targeting Android with your app, you do not need to do any more. But we will also try out the Windows desktop app support.

To build desktop apps for Windows, you must enable developer mode:

1. Navigate to **Start | Settings | Privacy & security | For developers**, and then switch on **Developer Mode**. (You can also search for “developers”.)
2. Accept the warning about how it “**could expose your device and personal data to security risk or harm your device.**”
3. Close the **Settings** app.

Creating a .NET MAUI project

We will now create a project for a cross-platform mobile and desktop app:

1. In Visual Studio, add a new project, as defined in the following list:
 - Project template: **.NET MAUI App** / `maui`

You can select **C#** for the language and **MAUI** for the project type to filter and show only the appropriate project templates.

- Project file and folder: `Northwind.Maui.Client`
- Solution file and folder: `ModernApps`

- **Framework: .NET 10.0 (Long Term Support)**
- **Include sample content:** Cleared
- **Enlist in Aspire orchestration:** Cleared

- On Windows, if you see a Windows security alert that **Windows Defender Firewall has blocked some features of Broker on all public and private networks**, then select **Private networks** and clear **Public networks**, and then click the **Allow access** button.
- In the project file, note the element that targets iOS, Android, and Mac Catalyst, the element to enable Windows targeting if the operating system is Windows, and the elements that set the project to be a single MAUI project, as shown highlighted in the following partial markup:

```
<Project Sdk="Microsoft.NET.Sdk">
<PropertyGroup> <TargetFrameworks>net10.0-
android;net10.0-ios; net10.0-maccatalyst</
TargetFrameworks> <TargetFrameworks
Condition="$([MSBuild]::IsOSPlatform('windows'))"
$(TargetFrameworks);net10.0-
windows10.0.19041.0 </TargetFrameworks>
... <RootNamespace>Northwind.Maui.Client</
RootNamespace> <UseMaui>true</UseMaui>
<SingleProject>true</SingleProject> ...
```

If you see the error Error NU1012 Platform version is not present for one or more target frameworks, even though they have specified a platform: net10.0-ios, net10.0-maccatalyst, then at the command prompt or terminal, in the

project folder, restore workloads for the project, as shown in the following command: `dotnet workload restore`

1. We are using CPM, so in the project file, remove the `version` attribute from any package references, as shown in the following markup:

```
<ItemGroup> <PackageReference
Include="Microsoft.Maui.Controls" />
<PackageReference
Include="Microsoft.Extensions.Logging.Debug"
/> </ItemGroup>
```

2. Build the project.
3. To the right of the **Run** button in the toolbar, set **Framework** to **net10.0-android**, and if necessary select the **Pixel 9 - API 36 (Android 16.0 - API 36)** emulator image that you previously created, as shown in *Figure 2.3*:

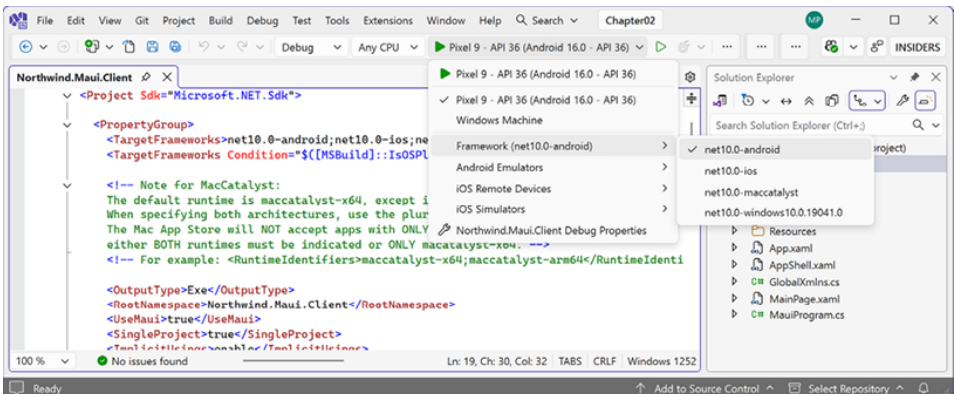


Figure 2.3: Selecting the Android emulator as the target for startup

1. Click the **Start** button in the Visual Studio toolbar and wait for the device emulator to start the Android operating system, and then deploy and

launch your mobile app. This can take more than five minutes, especially the first time that you build and start a new MAUI project. Keep an eye on the Visual Studio status bar, as shown in *Figure 2.4*:

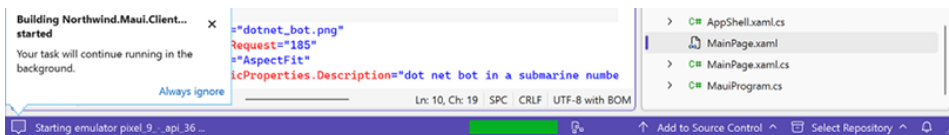


Figure 2.4: Status bar shows progress of the .NET MAUI app deployment

If you're doing this for the first time, there might be another Google license agreement to accept.

1. In the .NET MAUI app, click the **Click me** button to increment the counter three times, as shown in *Figure 2.5*:

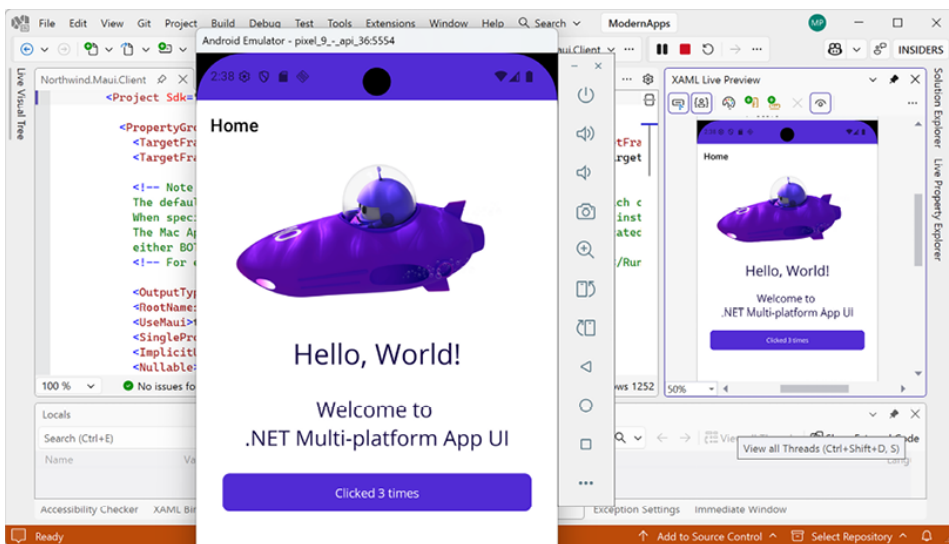


Figure 2.5: Incrementing the counter three times in the .NET MAUI app on Android

1. Close the Android device emulator by clicking the **x** in the top-right corner of the floating toolbar. You do not need to power down the emulator.

2. To the right of the **Start** button in the Visual Studio toolbar, set **Framework** to **net10.0-windows10.0.19041.0**.
3. Make sure that the **Debug** configuration is selected and then click the solid green triangle **Start** button labeled **Windows Machine**. You might see a warning about missing packages that should be installed on the first run. Just click the **Start** button a second time and they should now be installed and it will work.
4. After a few moments, note that the Windows app displays with the same **Click me** button and counter functionality as shown in *Figure 2.6*:

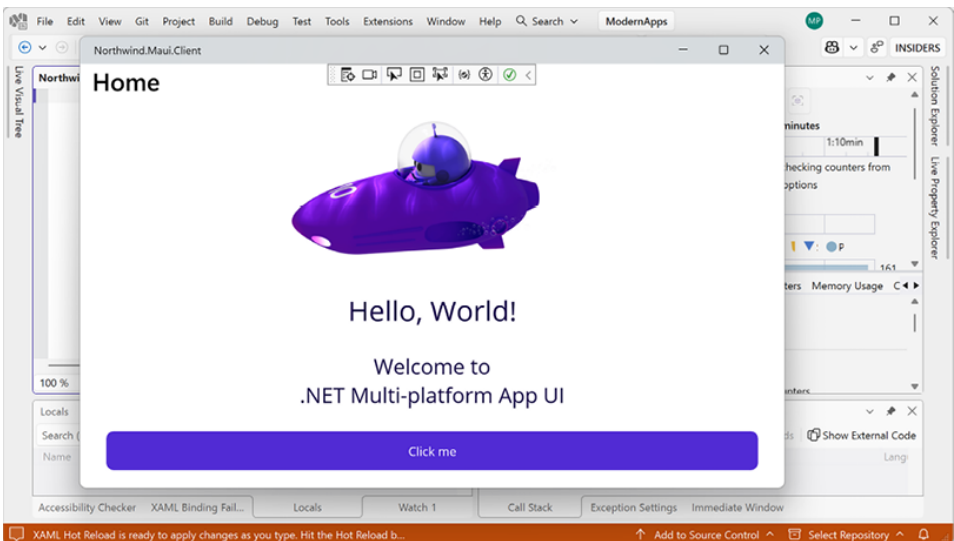


Figure 2.6: The same .NET MAUI app on Windows

1. Close the Windows app.

Good practice: You should test your .NET MAUI app on all the potential devices that it will need to run on. In this chapter, even if I do not explicitly tell you to do so, I recommend that you try the app by running it on your emulated Android device and on

Windows after each task to add a new feature. That way, you will have at least seen how it looks on a mobile device with a primarily tall and thin portrait size, and on a desktop device with a larger landscape size. If you are using a Mac, then I recommend that you test it in the iOS Simulator, Android Emulator, and as a Mac Catalyst desktop app.

Adding shell navigation and more content pages

Now, let's review the existing structure of the .NET MAUI app and then add some new pages and navigation to the project:

1. In the `Northwind.Maui.Client` project, in `MauiProgram.cs`, note that the `builder` object calls `UseMauiApp` and specifies `App` as its generic type, which is required for a .NET MAUI app, as shown highlighted in the following code:

```
using Microsoft.Extensions.Logging;
namespace Northwind.Maui.Client; public
static class MauiProgram { public static
MauiApp CreateMauiApp() { var builder =
MauiApp.CreateBuilder(); builder
.UseMauiApp<App>() .ConfigureFonts(fonts
=> { fonts.AddFont("OpenSans-Regular.ttf",
"OpenSansRegular");
fonts.AddFont("OpenSans-Semibold.ttf",
"OpenSansSemibold"); }); #if DEBUG
builder.Logging.AddDebug(); #endif return
builder.Build(); } }
```


2. In **Solution Explorer**, expand `App.xaml`, open `App.xaml.cs`, and note the `MainPage` property of the `App` is set to an instance of `AppShell`, which provides the shell of the user interface for your app, as shown highlighted in the following code:

```
namespace Northwind.Maui.Client; public
partial class App : Application { public
App() { InitializeComponent(); MainPage =
new AppShell(); } }
```

3. In `AppShell.xaml`, note that the shell initially has only a single content page named `MainPage`, as shown highlighted in the following code:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Shell
  x:Class="Northwind.Maui.Client.AppShell"
  xmlns="http://schemas.microsoft.com/
dotnet/2021/maui" xmlns:x="http://
schemas.microsoft.com/winfx/2009/xaml"
  xmlns:local="clr-
namespace:Northwind.Maui.Client"
  Title="Northwind.Maui.Client">
  <ShellContent Title="Home"
    ContentTemplate="{DataTemplate
local:MainPage}" Route="MainPage" /> </
Shell>
```

A shell with only one content page does not show any navigation because there is nowhere else to navigate to. You must have

at least two shell content items to see the navigation in the user interface.

1. In the `Resources` folder, in the `Images` folder, add images for some icons that we will use for flyout items in the navigation we are about to add.

You can download the images from the GitHub repository at the following link:

<https://github.com/markjprice/apps-services-net10/tree/main/code/Chapter02/Northwind.Maui.Client/Resources/Images>.

1. In `AppShell.xaml`, enable flyout mode, set the background to a pale blue color, add an icon for the `MainPage` content, add a flyout header, and then add some flyout items with more shell content, as shown highlighted in the following markup:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Shell
  x:Class="Northwind.Maui.Client.AppShell"
  xmlns="http://schemas.microsoft.com/dotnet/2021/maui" xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  xmlns:local="clr-namespace:Northwind.Maui.Client"
  Shell.FlyoutBehavior="Flyout"
  Title="Northwind.Maui.Client"
  FlyoutBackgroundColor="AliceBlue">
  <Shell.FlyoutHeader>
    <HorizontalStackLayout Spacing="10"
```

```

HorizontalOptions="Start"> <Image
Source="wind_face_3d.png"
WidthRequest="80" HeightRequest="80" />
<Label Text="Northwind"
FontFamily="OpenSansSemibold"
FontSize="32" VerticalOptions="Center" />
</HorizontalStackLayout> </
Shell.FlyoutHeader> <ShellContent
Title="Home" Icon="file_cabinet_3d.png"
ContentTemplate="{DataTemplate
local:MainPage}" Route="MainPage" />
<ShellContent Title="Categories"
Icon="delivery_truck_3d.png"
ContentTemplate="{DataTemplate
local:CategoriesPage}" Route="Categories"
/> <ShellContent Title="Products"
Icon="cityscape_3d.png"
ContentTemplate="{DataTemplate
local:ProductsPage}" Route="Products" />
<ShellContent Title="Customers"
Icon="card_index_3d.png"
ContentTemplate="{DataTemplate
local:CustomersPage}" Route="Customers" />
<ShellContent Title="Employees"
Icon="identification_card_3d.png"
ContentTemplate="{DataTemplate
local:EmployeesPage}" Route="Employees" />
<ShellContent Title="Settings"
Icon="gear_3d.png"
ContentTemplate="{DataTemplate
local:SettingsPage}" Route="Settings" />
</Shell>

```

You will see warnings on some of the
ContentTemplate lines about missing

pages because we have not created them yet. AliceBlue looks good in light mode, but if your operating system uses dark mode, then you might prefer an alternative color like #75858a.

1. In Visual Studio, right-click the `Northwind.Maui.Client` project folder, choose **Add | New Item...** or press *Ctrl + Shift + A*, select **.NET MAUI** in the template types tree, select **.NET MAUI ContentPage (XAML)**, enter the name `SettingsPage`, and click **Add**.

VS Code and Rider do not have project item templates for MAUI. You can create this item using the CLI, as shown in the following command:

```
dotnet new maui-page-xaml --name
SettingsPage.xaml
```

1. Repeat the previous step to add content pages named:
 - `CategoriesPage`
 - `CustomersPage`
 - `CustomerDetailPage`
 - `EmployeesPage`
 - `ProductsPage`
7. In **Solution Explorer**, double-click on the `CategoriesPage.xaml` file to open it for editing. Note that Visual Studio does not yet have a graphical design view for XAML.
8. In the `<ContentPage>` element, change the `Title` to `Categories`, and in the `<Label>` element, change the `Text` to

Categories, as shown highlighted in the following markup:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://
schemas.microsoft.com/dotnet/2021/maui"
xmlns:x="http://schemas.microsoft.com/
winfx/2009/xaml"
x:Class="Northwind.Maui.Client.CategoriesPage"
Title="Categories"> <VerticalStackLayout>
<Label Text="Categories"
VerticalOptions="Center"
HorizontalOptions="Center" /> </
VerticalStackLayout> </ContentPage>
```

9. Navigate to **View | Toolbox** or press *Ctrl + W, X*. Note that the toolbox has sections for **Controls**, **Layouts**, **Cells**, and **General**. If you are using a code editor without a toolbox, you can just type the markup instead of using the toolbox.
10. At the top of the toolbox is a search box. Enter the letter **b**, and then note that the list of controls is filtered to show controls like **Button**, **ProgressBar**, and **AbsoluteLayout**.
11. Drag and drop the **Button** control from the toolbox into the XAML markup after the existing `<Label>` control, before the closing element of the `VerticalStackLayout`, and change its `Text` property to `Hello!`, as shown in the following markup:

```
<Button Text="Hello!" />
```

12. Set the startup to **Windows Machine** and then start the `Northwind.Maui.Client` project with debugging. Note that the Visual Studio status bar shows us that **XAML Hot Reload** is connected.
13. In the top-left corner of the app, click the flyout menu (the “hamburger”

icon), and note the header and the images used for the icons in the flyout items, as shown in *Figure 2.7*:

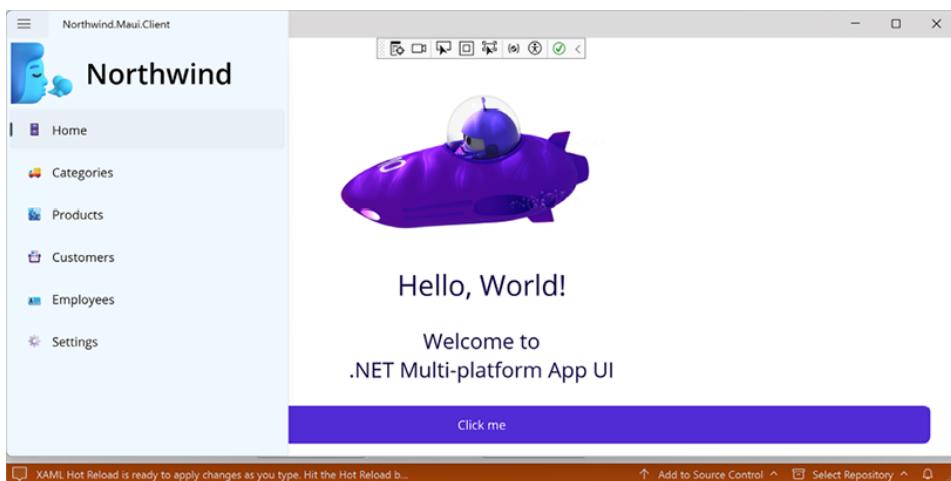


Figure 2.7: A flyout with image icons

1. In the flyout menu, click **Categories**, and note that the text on the button says **Hello!** and that it stretches across the width of the app window.
2. Leave the app running, and then in Visual Studio, change the `Text` property to `Click Me`, add an attribute to set the `WidthRequest` property to `100`, and note that the **XAML Hot Reload** feature automatically reflects the changes in the app itself, as shown in *Figure 2.8*:

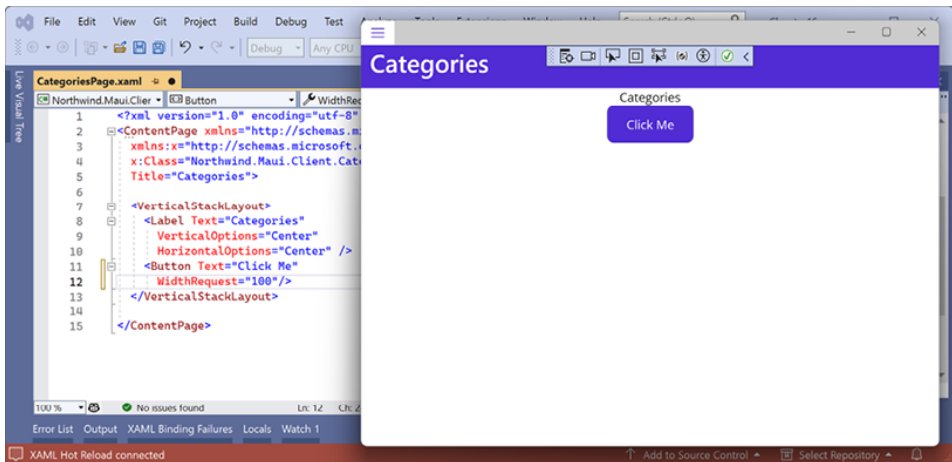


Figure 2.8: XAML Hot Reload automatically updating changes in the XAML in the live app

1. Close the app.
2. Modify the `Button` element to give it the name of `ClickMeButton` and a new event handler for its `Clicked` event, as shown in Figure 2.9:

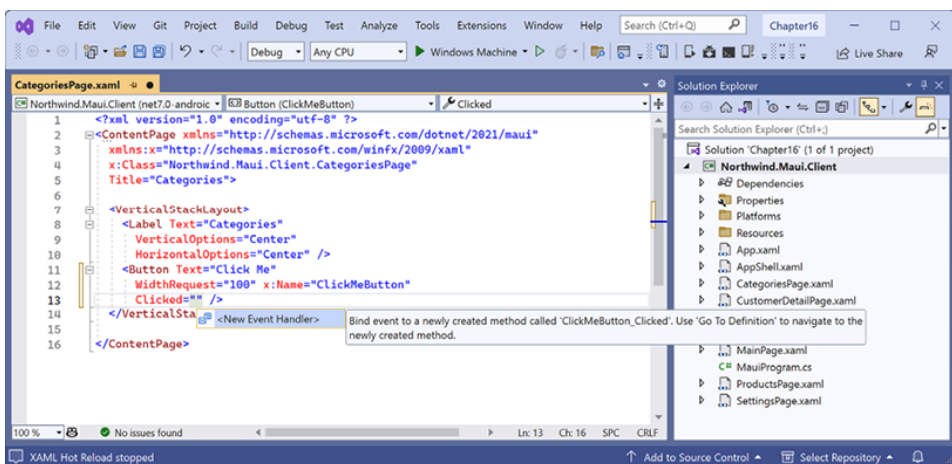


Figure 2.9: Adding an event handler to a control

1. Right-click the event handler name and select **Go To Definition** or press `F12`.
2. Add a statement to the event handler method that sets the content of the button to the current time, as shown highlighted in the following code:

```
private void ClickMeButton_Click(object  
sender, EventArgs e) { ClickMeButton.Text  
= DateTime.Now.ToString("hh:mm:ss"); }
```

3. Start the `Northwind.Maui.Client` project on at least one mobile device and one desktop device.

Good practice: When deploying to an Android emulator or an iOS simulator, the old version of the app could still be running. Make sure to wait for the deployment of the new version of your app before interacting with it. You can keep an eye on the Visual Studio status bar to track the deployment progress or just wait until you see the message **XAML Hot Reload connected**.

1. Navigate to **Categories**, click the button, and note that its text label changes to the current time.
2. Close the app.

Implementing more content pages

Now, let's implement some of the new pages:

1. In `EmployeesPage.xaml`, change the `Title` to `Employees` and add markup to define the UI for a simple calculator, as shown highlighted in the following markup:

```
<?xml version="1.0" encoding="utf-8" ?>  
<ContentPage xmlns="http://  
schemas.microsoft.com/dotnet/2021/maui"
```



```

xmlns:x="http://schemas.microsoft.com/
winfx/2009/xaml"
x:Class="Northwind.Maui.Client.EmployeesPage"
Title="Employees"> <VerticalStackLayout>
<Grid Background="DarkGray" Margin="10"
Padding="5" x:Name="GridCalculator"
ColumnDefinitions="Auto,Auto,Auto,Auto"
RowDefinitions="Auto,Auto,Auto,Auto">
<Button Grid.Row="0" Grid.Column="0"
Text="X" /> <Button Grid.Row="0"
Grid.Column="1" Text="/" /> <Button
Grid.Row="0" Grid.Column="2" Text="+" />
<Button Grid.Row="0" Grid.Column="3"
Text="-" /> <Button Grid.Row="1"
Grid.Column="0" Text="7" /> <Button
Grid.Row="1" Grid.Column="1" Text="8" />
<Button Grid.Row="1" Grid.Column="2"
Text="9" /> <Button Grid.Row="1"
Grid.Column="3" Text="0" /> <Button
Grid.Row="2" Grid.Column="0" Text="4" />
<Button Grid.Row="2" Grid.Column="1"
Text="5" /> <Button Grid.Row="2"
Grid.Column="2" Text="6" /> <Button
Grid.Row="2" Grid.Column="3" Text="." />
<Button Grid.Row="3" Grid.Column="0"
Text="1" /> <Button Grid.Row="3"
Grid.Column="1" Text="2" /> <Button
Grid.Row="3" Grid.Column="2" Text="3" />
<Button Grid.Row="3" Grid.Column="3"
Text="=" /> </Grid> <Label x:Name="Output"
FontSize="24" VerticalOptions="Center"
HorizontalOptions="Start" /> </
VerticalStackLayout> </ContentPage>

```

2. Add an event handler for the page's Loaded event, as shown

highlighted in the following markup:

```
Title="Employees"  
Loaded="ContentPage_Loaded">
```

3. In `EmployeesPage.xaml.cs`, add statements to resize each button in the grid and hook up an event handler for the `Clicked` event, as shown in the following code:

```
private void ContentPage_Loaded(object  
sender, EventArgs e) { foreach (Button  
button in  
GridCalculator.Children.OfType<Button>())  
{ button.FontSize = 24;  
button.WidthRequest = 54;  
button.HeightRequest = 54; button.Clicked  
+= Button_Clicked; } }
```

4. Add a `Button_Clicked` method with statements to handle the clicked button by concatenating the text of the button to the output label, as shown in the following code:

```
private void Button_Clicked(object sender,  
EventArgs e) { string operationChars =  
"+-/*X="; Button button = (Button)sender;  
if (operationChars.Contains(button.Text))  
{ Output.Text = string.Empty; } else {  
Output.Text += button.Text; } }
```

This is not a proper implementation for a calculator because the operations have not

been implemented. It just simulates one for now because we are focusing on how to build UIs with .NET MAUI. You can Google how to implement a simple calculator as an optional exercise.

Trying out the app

Now we can try out the calculator page in the app:

1. Start the `Northwind.Mauai.Client` project with at least one desktop and one mobile device.
2. Navigate to **Employees**, click some of the buttons, and note that the label updates to show what is clicked, as shown on an emulated Android device in *Figure 2.10*:

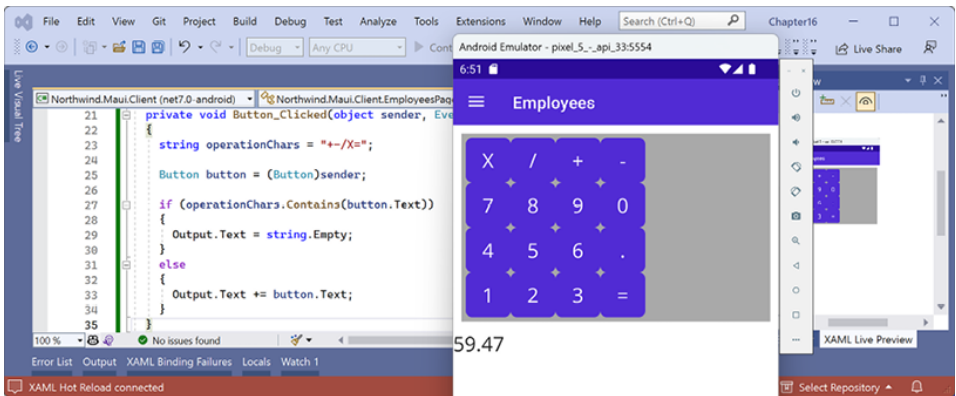


Figure 2.10: A simulated calculator on an emulated Android

1. Close the app.

By now, we have built a few simple UIs using XAML and MAUI. Next, let's see some techniques to improve apps, like defining and sharing resources.

Using shared resources

When building graphical UIs, you will often want to use a resource, such as a brush to paint the background of controls or an instance of a class to perform custom conversions. Resources can be defined at the following levels and shared with everything at that level or lower:

- Application
- Page
- Control

Defining resources to share across an app

A good place to define shared resources is at the app level, so let's see how to do that:

1. In the `Resources` folder, in the `Styles` folder, add a new **.NET MAUI Resource Dictionary (XAML)** project item named `Northwind.xaml`.

VS Code and Rider do not have project item templates for MAUI. You can create this item using the CLI, as shown in the following command:

```
dotnet new maui-dict-  
xaml --name  
Northwind.xaml
```

1. Add markup inside the existing `ResourceDictionary` element to define a linear gradient brush with a key of `Rainbow`, as shown highlighted in the following markup:

```
<?xml version="1.0" encoding="utf-8" ?>
<ResourceDictionary xmlns="http://
schemas.microsoft.com/dotnet/2021/maui"
xmlns:x="http://schemas.microsoft.com/
winfx/2009/xaml"
x:Class="Northwind.Maui.Client.Resources.Styles.N
<LinearGradientBrush x:Key="Rainbow">
<GradientStop Color="Red" Offset="0" />
<GradientStop Color="Orange" Offset="0.1"
/> <GradientStop Color="Yellow"
Offset="0.3" /> <GradientStop
Color="Green" Offset="0.5" />
<GradientStop Color="Blue" Offset="0.7" />
<GradientStop Color="Indigo" Offset="0.9"
/> <GradientStop Color="Violet" Offset="1"
/> </LinearGradientBrush> </
ResourceDictionary>
```

2. In App.xaml, add an entry to the merged resource dictionaries to reference the resource file in the Styles folder named Northwind.xaml, as shown highlighted in the following markup:

```
<?xml version = "1.0" encoding = "UTF-8" ?
> <Application xmlns="http://
schemas.microsoft.com/dotnet/2021/maui"
xmlns:x="http://schemas.microsoft.com/
winfx/2009/xaml" xmlns:local="clr-
namespace:Northwind.Maui.Client"
x:Class="Northwind.Maui.Client.App">
<Application.Resources>
<ResourceDictionary>
<ResourceDictionary.MergedDictionaries>
<ResourceDictionary Source="Resources/
Styles/Colors.xaml" /> <ResourceDictionary
```

```
Source="Resources/Styles/Styles.xaml" />
<ResourceDictionary Source="Resources/
Styles/Northwind.xaml" /> </
ResourceDictionary.MergedDictionaries> </
ResourceDictionary> </
Application.Resources> </Application>
```

Referencing shared resources

Now we can reference the application-level shared resource:

1. In `CategoriesPage.xaml`, modify the `ContentPage` to set its background to the brush resource with the key of `Rainbow`, as shown highlighted in the following markup:

```
<ContentPage xmlns="http://
schemas.microsoft.com/dotnet/2021/maui"
xmlns:x="http://schemas.microsoft.com/
winfx/2009/xaml"
x:Class="Northwind.Maui.Client.CategoriesPage"
Background="{StaticResource Rainbow}"
Title="Categories">
```

`StaticResource` means the resource is read once when the app first starts. If the resource changes after that, any elements that reference it will not be updated.

1. Start the `Northwind.Maui.Client` project with debugging.
2. Navigate to **Categories** and note that the background of the page is a rainbow.
3. Close the app.

Changing shared resources dynamically

Now we can implement a settings page to allow the user to change between light mode, dark mode, or system mode used in the UI at runtime:

1. In the `Resources` folder, in the `Styles` folder, add a new **.NET MAUI Resource Dictionary (XAML)** project item named `LightDarkModeColors.xaml`.
2. Add markup inside the existing `ResourceDictionary` element to define sets of suitable colors for light and dark mode, as shown in the following markup:

```
<?xml version="1.0" encoding="utf-8" ?>
<ResourceDictionary xmlns="http://
schemas.microsoft.com/dotnet/2021/maui"
xmlns:x="http://schemas.microsoft.com/
winfx/2009/xaml"
x:Class="Northwind.Maui.Client.Resources
.Styles.LightDarkModeColors"> <Color
x:Key="LightPageBackgroundColor">White</
Color> <Color
x:Key="LightNavigationBarColor">AliceBlue</
Color> <Color
x:Key="LightPrimaryColor">WhiteSmoke</
Color> <Color
x:Key="LightSecondaryColor">Black</Color>
<Color
x:Key="LightPrimaryTextColor">Black</
Color> <Color
x:Key="LightSecondaryTextColor">White</
Color> <Color
x:Key="LightTertiaryTextColor">Gray</
Color> <Color
x:Key="DarkPageBackgroundColor">Black</
Color> <Color
```

```

x:Key="DarkNavigationBarColor">Teal</
Color> <Color
x:Key="DarkPrimaryColor">Teal</Color>
<Color x:Key="DarkSecondaryColor">White</
Color> <Color
x:Key="DarkPrimaryTextColor">White</Color>
<Color
x:Key="DarkSecondaryTextColor">White</
Color> <Color
x:Key="DarkTertiaryTextColor">WhiteSmoke</
Color> </ResourceDictionary>

```

3. In the Resources folder, in the Styles folder, add a new **.NET MAUI Resource Dictionary (XAML)** project item named `DarkModeTheme.xaml`.
4. Add markup inside the existing `ResourceDictionary` element to define styles to use in dark mode, as shown in the following markup:

```

<?xml version="1.0" encoding="utf-8" ?>
<ResourceDictionary xmlns="http://
schemas.microsoft.com/dotnet/2021/maui"
xmlns:x="http://schemas.microsoft.com/
winfx/2009/xaml"
x:Class="Northwind.Maui.Client.Resources.Styles.D
<Style TargetType="Shell"> <Setter
Property="FlyoutBackgroundColor"
Value="{StaticResource
DarkNavigationBarColor}" /> </Style>
<Style TargetType="ContentPage"> <Setter
Property="BackgroundColor"
Value="{StaticResource
DarkPageBackgroundColor}" /> </Style>
<Style TargetType="Button"> <Setter
Property="BackgroundColor"

```



```

Value="{StaticResource DarkPrimaryColor}"
/> <Setter Property="TextColor"
Value="{StaticResource
DarkSecondaryColor}" /> <Setter
Property="HeightRequest" Value="45" />
<Setter Property="WidthRequest"
Value="190" /> <Setter
Property="CornerRadius" Value="18" /> </
Style> </ResourceDictionary>

```

5. In the Resources folder, in the Styles folder, add a new **.NET MAUI Resource Dictionary (XAML)** project item named `LightModeTheme.xaml`.
6. Add markup inside the existing `ResourceDictionary` element to define styles to use in light mode, as shown in the following markup:

```

<?xml version="1.0" encoding="utf-8" ?>
<ResourceDictionary xmlns="http://
schemas.microsoft.com/dotnet/2021/maui"
xmlns:x="http://schemas.microsoft.com/
winfx/2009/xaml"
x:Class="Northwind.Maui.Client.Resources.Styles.L
<Style TargetType="Shell"> <Setter
Property="FlyoutBackgroundColor"
Value="{StaticResource
LightNavigationBarColor}" /> </Style>
<Style TargetType="ContentPage"> <Setter
Property="BackgroundColor"
Value="{StaticResource
LightPageBackgroundColor}" /> </Style>
<Style TargetType="Button"> <Setter
Property="BackgroundColor"
Value="{StaticResource LightPrimaryColor}"
/> <Setter Property="TextColor"

```

```

Value="{StaticResource
LightSecondaryColor}" /> <Setter
Property="HeightRequest" Value="45" />
<Setter Property="WidthRequest"
Value="190" /> <Setter
Property="CornerRadius" Value="18" /> </
Style> </ResourceDictionary>

```

7. In the Resources folder, in the Styles folder, add a new **.NET MAUI Resource Dictionary (XAML)** project item named `SystemModeTheme.xaml`.
8. Add markup inside the existing `ResourceDictionary` element to define styles to use with light and dark mode depending on how the operating system has had its option set, as shown in the following markup:

```

<?xml version="1.0" encoding="utf-8" ?>
<ResourceDictionary xmlns="http://
schemas.microsoft.com/dotnet/2021/maui"
xmlns:x="http://schemas.microsoft.com/
winfx/2009/xaml"
x:Class="Northwind.Maui.Client.Resources.Styles.S
<Style TargetType="Shell"> <Setter
Property="FlyoutBackgroundColor"
Value="{AppThemeBinding
Light={StaticResource
LightNavigationBarColor},
Dark={StaticResource
DarkNavigationBarColor}}" /> </Style>
<Style TargetType="ContentPage"> <Setter
Property="BackgroundColor"
Value="{AppThemeBinding
Light={StaticResource
LightPageBackgroundColor},

```

```

Dark={StaticResource
DarkPageBackgroundColor}}" /> </Style>
<Style TargetType="Button"> <Setter
Property="BackgroundColor"
Value="{AppThemeBinding
Light={StaticResource LightPrimaryColor},
Dark={StaticResource DarkPrimaryColor}}" /
> <Setter Property="TextColor"
Value="{AppThemeBinding
Light={StaticResource
LightSecondaryColor}, Dark={StaticResource
DarkSecondaryColor}}" /> <Setter
Property="HeightRequest" Value="45" />
<Setter Property="WidthRequest"
Value="190" /> <Setter
Property="CornerRadius" Value="18" /> </
Style> </ResourceDictionary>

```

Note the use of the `AppThemeBinding` extension to dynamically bind to two pre-defined special filters, `Light` and `Dark`. These are bound to system modes.

1. In `App.xaml`, add the light and dark mode colors and system theme resources, as shown highlighted in the following markup:

```

<ResourceDictionary.MergedDictionaries>
<ResourceDictionary Source="Resources/
Styles/Colors.xaml" /> <ResourceDictionary
Source="Resources/Styles/Styles.xaml" />
<ResourceDictionary Source="Resources/
Styles/Northwind.xaml" />
<ResourceDictionary Source= "Resources/
Styles/LightDarkModeColors.xaml" />

```

```
<ResourceDictionary Source= "Resources/  
Styles/SystemModeTheme.xaml" /> </  
ResourceDictionary.MergedDictionaries>
```

Good practice: The

SystemModeTheme.xaml resources file references colors defined in the LightDarkModelColors.xaml file, so the order is important.

1. In the project, create a folder named `Controls`.
2. In the `Controls` folder, add a class file named `ThemeEnum.cs`, and define an `enum` type with three values: `System`, `Light`, and `Dark`, for selecting themes, as shown in the following code:

```
namespace Northwind.Maui.Client.Controls;  
public enum Theme { System, Light, Dark }
```

3. In the `Controls` folder, add a class file named `EnumPicker.cs`, and define a class that inherits from the `Picker` control that can be bound to any `enum` type and show a dropdown list of its values, as shown in the following code:

```
using System.Reflection; // To use  
GetTypeInfo method. namespace  
Northwind.Maui.Client.Controls; public  
class EnumPicker : Picker { public Type  
EnumType { set =>  
SetValue(EnumTypeProperty, value); get =>  
(Type)GetValue(EnumTypeProperty); } public  
static readonly BindableProperty  
EnumTypeProperty =
```

```
BindableProperty.Create( propertyName:
nameof(EnumType), returnType:
typeof(Type), declaringType:
typeof(EnumPicker), propertyChanged:
(bindable, oldValue, newValue) => {
EnumPicker picker = (EnumPicker)bindable;
if (oldValue != null) { picker.ItemsSource
= null; } if (newValue != null) { if (!
((Type)newValue).GetTypeInfo().IsEnum)
throw new ArgumentException( "EnumPicker:
EnumType property must be enumeration
type"); picker.ItemsSource =
Enum.GetValues((Type)newValue); } }); }
```

4. In `SettingsPage.xaml`, import a local namespace for using our custom `EnumPicker` control, an `ios` namespace for adding a special property that only applies to iOS apps, change the `Title` to `Settings`, and create an instance of `EnumPicker` for selecting a theme, as shown highlighted in the following markup:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://
schemas.microsoft.com/dotnet/2021/maui"
xmlns:x="http://schemas.microsoft.com/
winfx/2009/xaml" xmlns:local="clr-
namespace:Northwind.Maui.Client.Controls"
xmlns:ios="clr-
namespace:Microsoft.Maui.Controls.PlatformConfigur
x:Class="Northwind.Maui.Client.SettingsPage"
Title="Settings"> <VerticalStackLayout
HorizontalOptions="Center">
<local:EnumPicker
ios:Picker.UpdateMode="WhenFinished"
EnumType="{x:Type local:Theme}"
```

```

Title="Select Theme"
SelectedIndexChanged="ThemePicker_SelectionChanged"
Loaded="ThemePicker_Loaded"
x:Name="ThemePicker" /> </
VerticalStackLayout> </ContentPage>

```

5. In `SettingsPage.xaml.cs`, add statements to handle the events for the picker, as shown highlighted in the following code:

```

using Northwind.Maui.Client.Controls; //
To use enum Theme. using
Northwind.Maui.Client.Resources.Styles; //
To use DarkModeTheme. namespace
Northwind.Maui.Client; public partial
class SettingsPage : ContentPage { public
SettingsPage() { InitializeComponent(); }
private void ThemePicker_SelectionChanged(
object sender, EventArgs e) { Picker
picker = sender as Picker; Theme theme =
(Theme)picker.SelectedItem;
ICollection<ResourceDictionary> resources
=
Application.Current.Resources.MergedDictionaries;
if (resources is not null) {
resources.Clear(); resources.Add(new
Resources.Styles.Northwind());
resources.Add(new LightDarkModeColors());
ResourceDictionary themeResource = theme
switch { Theme.Dark => new
DarkModeTheme(), Theme.Light => new
LightModeTheme(), _ => new
SystemModeTheme() };
resources.Add(themeResource); } } private
void ThemePicker_Loaded(object sender,

```

```
EventArgs e) { ThemePicker.SelectedItem =  
Theme.System; } }
```

6. Start the `Northwind.Maui.Client` project with at least one desktop and mobile device.
7. Note that the colors and shape of the button on the home page are in light mode, as shown on a **Windows** machine in *Figure 2.11*:

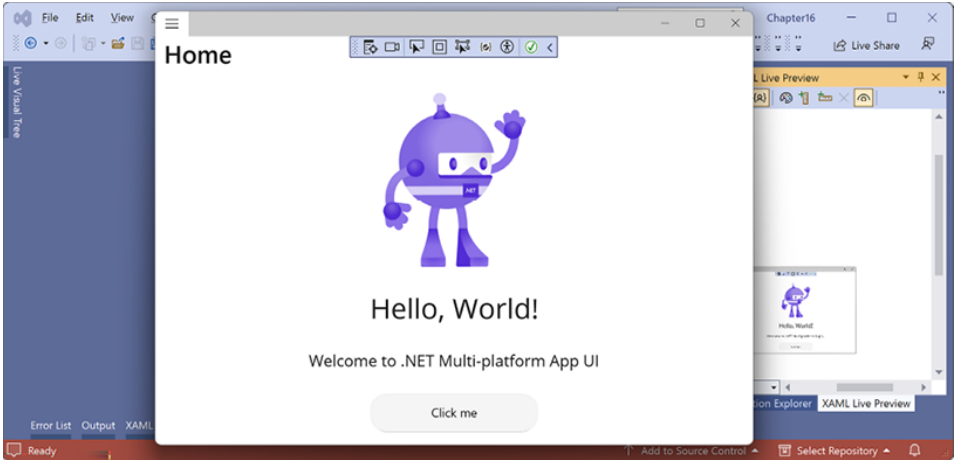


Figure 2.11: Light mode button on Windows

1. Leave the app running, start the Windows **Settings** app, navigate to **Personalization | Colors**, and in the **Choose your mode** section, select **Dark**, as shown in *Figure 2.12*:

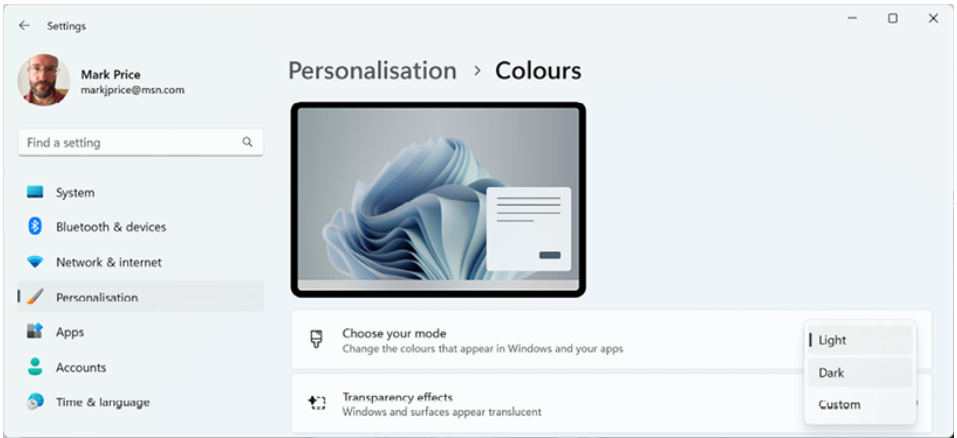


Figure 2.12: Switching the system color mode in Windows Settings

1. Leave **Settings** open, switch back to the app, and note that it has dynamically switched to dark mode colors, as shown in *Figure 2.13*:

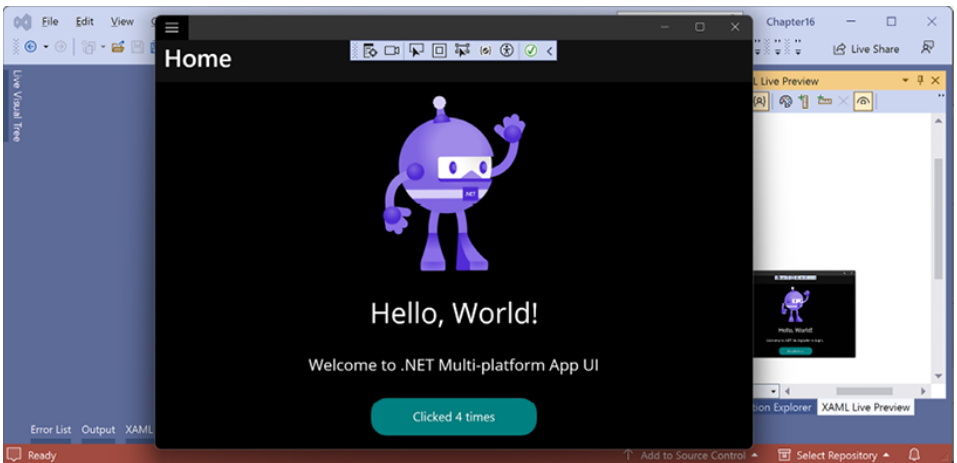


Figure 2.13: Dark mode in our app

1. Close the app.
2. In **Settings**, switch back to **Light** mode.

Good practice: A resource can be an instance of any object. To share it within an application, define

it in the `App.xaml` file and give it a unique key. To set an element's property with a resource once when the app first starts, use `{StaticResource key}`. To set an element's property with a resource whenever the resource value changes during the lifetime of the app, use `{DynamicResource key}`. To load a resource using code, use the `TryGetValue` method of the `Resources` property. If you treat the `Resources` property as a dictionary and use array-style syntax, like `Resources[key]`, it will only find resources defined directly in the dictionary, not in any merged dictionaries.

Resources can be defined and stored inside any element of XAML, not just at the app level. For example, if a resource is only needed on `MainPage`, then it can be defined there. You can also dynamically load XAML files at runtime.

You can read more about .NET MAUI resource dictionaries at the following link: <https://learn.microsoft.com/en-us/dotnet/maui/fundamentals/resource-dictionaries>. In particular, note the section about resource lookup behavior.

Introducing data binding

When building graphical UIs, you will often want to bind a property of one control to another, or to some data.

The simplest type of binding is between two elements. One element acts as a source for a value and the other elements acts as the target.

Let's take the following steps:

1. In `CategoriesPage.xaml`, under the existing button in the vertical stack layout, add a label for instructions, another label to show the current degree of rotation, a slider for selecting a rotation, and a rainbow square to rotate, as shown highlighted in the following markup:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://
schemas.microsoft.com/dotnet/2021/maui"
xmlns:x="http://schemas.microsoft.com/
winfx/2009/xaml"
x:Class="Northwind.Maui.Client.CategoriesPage"
Background="{StaticResource Rainbow}"
Title="Categories"> <VerticalStackLayout>
<Button Text="Click Me" WidthRequest="100"
x:Name="ClickMeButton"
Clicked="ClickMeButton_Clicked" /> <Label
Margin="10">Use the slider to rotate the
square:</Label> <Label
BindingContext="{x:Reference
Name=SliderRotation}" Text="{Binding
Path=Value, StringFormat='{0:N0}
degrees'}" FontSize="30"
HorizontalTextAlignment="Center" />
<Slider Value="0" Minimum="0"
Maximum="180" x:Name="SliderRotation"
Margin="10,0" /> <Rectangle
HeightRequest="200" WidthRequest="200"
Fill="{StaticResource Rainbow}"
BindingContext="{x:Reference
Name=SliderRotation}" Rotation="{Binding
Path=Value}" /> </VerticalStackLayout> </
ContentPage>
```

Note that the text of the label and the angle

of the rotation of the rectangle are both bound to the slider's value using a binding context and the `{Binding}` markup extension.

1. Start the `Northwind.Maui.Client` project.
2. Navigate to the **Categories** page.
3. Click and pull the slider to change the rotation of the rainbow square, as shown in *Figure 2.14*:

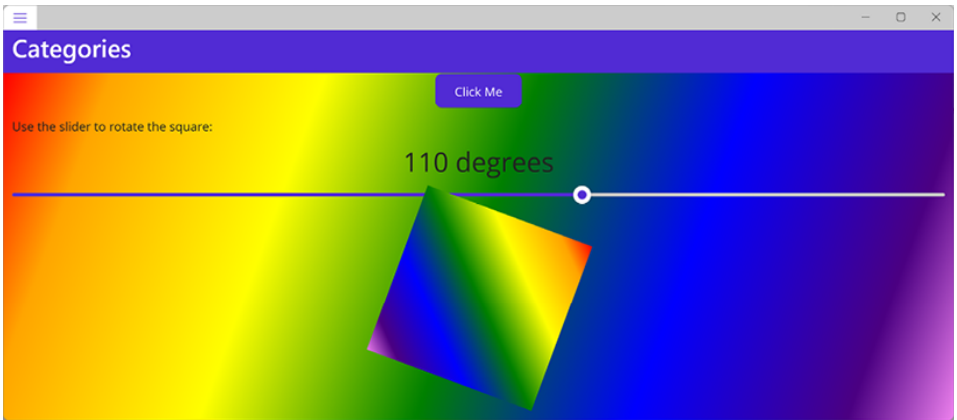


Figure 2.14: A slider data bound to a label and the rotation of a rectangle on Windows

1. Close the app.

Other common types of binding are covered in the optional exercise about implementing MVVM in the next section.

Practicing and exploring

Test your knowledge and understanding by answering some questions, getting some hands-on practice, and exploring this chapter's topics with more in-depth research.

Exercise 2.1 – Test your knowledge

Answer the following questions:

1. What are the four categories of .NET MAUI UI components, and what do they represent?
2. What is the benefit of the `Shell` component and what kinds of UI does it implement?
3. How can you enable a user to perform an action on a cell in a list view?
4. When would you use an `Entry` instead of an `Editor`?
5. What is the effect of setting `IsDestructive` to `true` for a menu item in a cell's context actions?
6. You have defined a `Shell` with a content page, but no navigation is shown. Why might this be?
7. What is the difference between `Margin` and `Padding` for an element like a `Button`?
8. How are event handlers attached to an object using XAML?
9. What do XAML styles do?
10. Where can you define resources?

Exercise 2.2 – Explore topics

Use the links on the following page to learn more detail about the topics covered in this chapter: <https://github.com/markjprice/apps-services-net10/blob/main/docs/book-links.md#chapter-2---building-mobile-apps-using-net-maui>.

Exercise 2.3 – Implementing Model-View-ViewModel for .NET MAUI

In this online-only section, you will learn how to implement the MVVM design pattern with a .NET MAUI app: <https://github.com/markjprice/apps-services-net10/blob/main/docs/ch16-mvvm.md>.

Exercise 2.4 – Integrating .NET MAUI apps with Blazor and native platforms

In this online-only section, you will learn how to integrate a .NET MAUI app with native mobile features: <https://github.com/markjprice/apps-services-net10/blob/main/docs/ch02-maui-blazor.md>.

Summary

In this chapter, you learned:

- How to build a cross-platform mobile and desktop app using .NET MAUI.
- How to define shared resources and reference them.
- How to use data binding with common controls.

In the next chapter, you will learn how to build desktop apps using Avalonia.

Get This Book **UNLOCK NOW** **ion and Exclusive**

Scan the QR code
book by name, co



Search for this
ps on the page.

Note: Keep your invoice handy. Purchases made directly from Packt don't require one.

3

Building Desktop Apps Using Avalonia

This chapter is about building desktop apps for Windows, macOS, and Linux using Avalonia. First, we will introduce Avalonia and how it is similar to and different from technologies like .NET MAUI. You will install the Avalonia project templates and set up your code editor to work with Avalonia, ready to create your own apps. Next, we will review the common controls for building a user interface, including those for layout, and how to style controls. Then we will create a simple desktop app with a menu and use assets like images.

This chapter will cover the following topics:

- Understanding Avalonia
- Working with Avalonia controls
- Creating an Avalonia project
- Referencing assets like images

Understanding Avalonia

Avalonia and .NET MAUI are both modern UI frameworks for cross-platform development in .NET, but they differ significantly in their architecture, rendering approach, and UI philosophy.

Avalonia has its own completely custom rendering engine, built from the ground up for cross-platform performance. Avalonia does not rely on native UI components; instead, it renders everything itself using Skia, an open-source 2D graphics library. Since it doesn't use native widgets, an Avalonia app looks the same regardless of OS, just like a web page would.

.NET MAUI uses native controls where possible, ensuring an app looks like other apps on the same platform, but falls back to drawing elements using Skia in some cases. On iOS, it wraps UIKit; on Android, it wraps Android Views; on Windows, it wraps WinUI 3. UI elements may behave differently on different operating systems. The native wrapper approach means more complexity and potential performance loss compared to direct rendering, and it takes Microsoft longer to support new OS features.

Due to these differences, Avalonia is stronger on desktop and embedded platforms, while .NET MAUI is stronger for mobile.

Avalonia is quietly dominating the HMI space:



Factory control panels



Medical device interfaces



Scientific instrument displays

The reason?

Direct GPU rendering on embedded Linux. No desktop environment needed, just beautiful, responsive UIs on minimal hardware.

Quoted from Avalonia tweet: <https://bsky.app/profile/avaloniaui.net/post/3lvuio237qh2j>

Avalonia is supported on the platforms listed at the

following link: <https://docs.avaloniaui.net/docs/overview/supported-platforms>.

Developer experience

Avalonia is similar to **Windows Presentation Foundation (WPF)**, so developers familiar with WPF or Silverlight will find Avalonia easy to use. Since it renders everything itself, you can fully customize how UI elements appear. But Avalonia has fewer third-party UI libraries and integrations compared to MAUI. Avalonia is open-source and community-driven, with a more cross-platform focus, including Linux.

.NET MAUI is the successor to Xamarin.Forms, so developers from that ecosystem will find it more natural. It offers better tools for mobile, including gestures, animations, and platform services. .NET MAUI has more enterprise support via Microsoft with a stronger mobile ecosystem due to Xamarin's legacy.

In *Chapter 2, Building Mobile Apps Using .NET MAUI*, you were introduced to XAML, a markup language for graphical user interfaces. In Avalonia, XAML files have the extension `.axaml` instead of `.xaml`. This represents **Avalonia XAML**, and the file extension was introduced for technical reasons to allow integration with Visual Studio.

If your target is desktop-first and you care about Linux, pixel-consistent rendering, and a very flexible styling model, Avalonia usually feels like the better fit. If you need mobile first-class support on iOS and Android with native controls, .NET MAUI makes sense.

Linux and WebAssembly targets

.NET MAUI's officially supported targets are Android, iOS, macOS, and Windows. Linux is not on that list. Avalonia ships Linux support out of the box, which matters if you want one codebase across Windows, macOS, and Linux.

If you want a desktop app that can also run in the browser without rewriting the UI, Avalonia supports WebAssembly. MAUI's official targets do not include WebAssembly.

Consistent, Skia-based rendering

Avalonia draws all controls with its own renderer, so the UI looks and behaves the same on every desktop OS. That cuts down on “works on Win, looks odd on Mac” bugs that come from native-control mismatches.

Tooling that's friendly to Rider and cross-OS devs

If you live in Rider, Avalonia has a first-party experience with a live XAML previewer. That's handy on Windows, macOS, or Linux dev machines.

Good practice: Choose Avalonia if you are building a cross-platform desktop app with full control over UI and performance. Choose .NET MAUI if you need mobile support or prefer using native UI components for platform consistency.

Skia graphics engine

Skia is an open-source 2D graphics library that provides a powerful and efficient way to render graphics across multiple platforms. It's used in several major applications, including Google Chrome, Flutter, Android, Firefox, and Avalonia.

Skia acts as a graphics engine that provides vector graphics rendering (lines, shapes, paths), text rendering, image rasterization and manipulation, hardware-accelerated rendering using OpenGL, Vulkan, and Metal, or software-based rendering for systems without GPU support. Instead of using native UI frameworks, Skia handles rendering from scratch, making it platform-agnostic.

I've compared Avalonia with .NET MAUI for user interface rendering in *Table*

3.1:

.NET MAUI (UI & Skia Native UI)			
Reliance on platform-specific UI			
Consistent look and feel across platforms			
Reliance on OS-specific optimizations			
Customizable UI for platform UI			

Table 3.1: Comparing Avalonia with .NET MAUI for UI rendering

To summarize, Avalonia uses Skia for rendering, allowing it to bypass native UI limitations, while .NET MAUI wraps native controls, making it more dependent on platform-specific rendering.

Warning! On Linux, Skia is built against glibc 2.17. If your distro uses something else instead, you need to build your own libSkiaSharp.so at SkiaSharp. You can also visit the SkiaSharp home page for more information about supported versions at the following link: <https://github.com/mono/SkiaSharp>.

While frameworks like .NET MAUI abstract over native UI controls, Avalonia takes a different approach of using Skia’s rendering engine to draw the user controls, as shown in *Figure 3.1*:

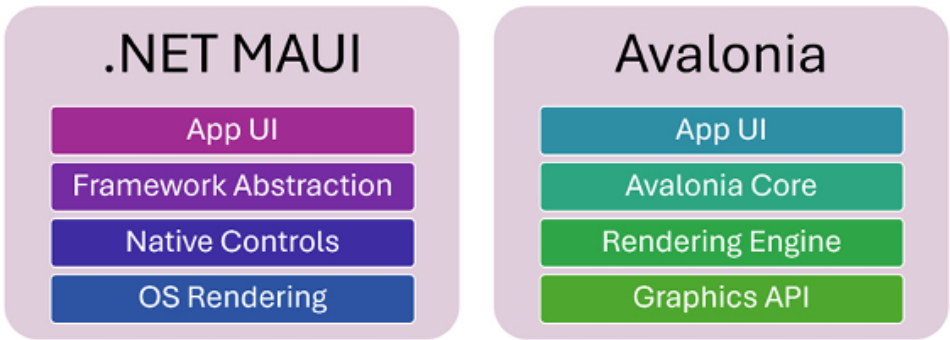


Figure 3.1: Architectural differences between .NET MAUI and Avalonia

This architectural difference provides several benefits:

- Consistent behavior across platforms
- Pixel-perfect rendering and full control over the UI stack
- Simplified platform support and reduced maintenance overhead
- Better performance on resource-constrained devices

Now that we have covered some theory about Avalonia, let's get practical and install the Avalonia project templates so that we can build a desktop app.

Installing the Avalonia project templates

To create Avalonia projects, you will first need to install the templates:

1. At a command prompt or terminal, install the Avalonia project templates, as shown in the following command:

```
dotnet new install Avalonia.Templates
```

2. Note the successful result, as shown in the following partial output (I removed the `Tags` column to save space):

```
Success: Avalonia.Templates::11.2.1.1
installed the following templates:
Template Name Short Name Language
-----
Avalonia .NET App avalonia.app [C#],F#
Avalonia .NET MVVM App avalonia.mvvm
[C#],F# Avalonia Cross Platform
Application avalonia.xplat [C#],F#
Avalonia Resource Dictionary
avalonia.resource Avalonia Styles
```

```
avalonia.styles Avalonia TemplatedControl  
avalonia.templatedcontrol [C#],F# Avalonia  
UserControl avalonia.usercontrol [C#],F#  
Avalonia Window avalonia.window [C#],F#
```

The first three are project templates. The last five are item templates that can only be created in a project folder.

The three Avalonia project templates differ mainly in scope and starter architecture:

- Use `avalonia.app` for the smallest, code-behind desktop app.
- Use `avalonia.mvvm` for a desktop app set up with MVVM out of the box.
- Use `avalonia.xplat` when you want one solution that targets desktop, plus optional iOS, Android, and WebAssembly from day one.

Avalonia .NET App (`avalonia.app`)

Summary of this template:

- Targets desktop only: Windows, macOS, Linux.
- Code-behind pattern by default. You still get XAML views, but no MVVM plumbing is added for you.
- Smallest project, fewest moving parts. Good for prototypes, tools, utilities, teaching demos, or if you prefer to add MVVM or DI yourself later.

Use this template when you want the leanest possible starter and you're fine driving with code-behind to begin with, or you're building a small internal tool, sample, or a quick prototype and don't want a lot of scaffolding yet. If you're experimenting or learning concepts, `avalonia.app` is uncluttered.

Avalonia .NET MVVM App (avalonia.mvvm)

Summary of this template:

- Targets desktop only: Windows, macOS, Linux.
- Prewired for MVVM out of the box. The official docs state MVVM is the recommended pattern, and the template is set up accordingly. Typically includes basic ViewModel scaffolding and data binding patterns so you can scale cleanly.

Use this template when desktop is your focus and you want a sane, production-friendly structure from day one, or you like a clean MVVM separation for testability, larger codebases, and theming. The docs recommend MVVM for Avalonia apps, so this is the “safe default” for most desktop apps.

We will use this Avalonia .NET MVVM App (avalonia.mvvm) project template in this chapter.

Avalonia Cross Platform Application (avalonia.xplat)

Summary of this template:

- Multi-target solution meant to add mobile and web on top of desktop. The docs call out “all supported platforms,” which covers Windows, macOS, Linux, iOS, Android, and WebAssembly.
- You’ll need the extra SDK workloads for mobile and web to build and run those apps. You still get a desktop app, but the project structure is set so you can enable iOS/Android/WASM later without rearranging your solution.

Use this template when you plan to ship desktop first, but you also want the option to add iOS/Android/Web later without reshaping the project structure.

Setting up your code editor for Avalonia

Avalonia supports Visual Studio, VS Code, and Rider for development.

Rider has built-in support for Avalonia XAML, including first-class support for Avalonia-specific XAML features and custom code inspections. Now that Rider is free for individual use, I recommend it as the primary IDE for Avalonia development, especially for developers on macOS and Linux.

Good practice: If you're developing Avalonia with Visual Studio, you should install the **Avalonia for Visual Studio** extension because it provides IntelliSense support for Avalonia XAML together with a previewer. You can search for it within Visual Studio by navigating to **Extensions | Manage Extensions**, or use the following link:

```
https://  
marketplace.visualstudio.com/  
items?  
itemName=AvaloniaTeam.AvaloniaVS.
```

The **Avalonia for VS Code Extension** contains basic support for Avalonia XAML autocomplete and a previewer, but the experience is not as rich as what you'll find in Rider or Visual Studio. You can install the extension from the VS Code marketplace at the following link: [https://
marketplace.visualstudio.com/items?
itemName=AvaloniaTeam.vscode-avalonia](https://marketplace.visualstudio.com/items?itemName=AvaloniaTeam.vscode-avalonia).

Now that you've been introduced to Avalonia and its project templates, you're ready to dive into the parts of the framework you'll spend most of your time with: the controls. Controls are the building blocks of every Avalonia application, from simple buttons and text boxes to more complex layouts and custom UI elements. In the next section, we'll look at how controls are structured, how to add them to your application, and how they can be styled to create a responsive and polished user experience.

Working with Avalonia controls

At this point, we should learn some basics about working with Avalonia controls: how to locate one in code, how to handle events, how to lay out controls with common properties, and how to style controls.

Locating controls

When working with code-behind, you often need to access the controls defined in XAML.

You first need to obtain a reference to the desired control. To do this, give the control a name using the `Name` (or `x:Name`) attribute in XAML. For example, a button named `greetingButton`, as shown in the following markup:

```
<Window xmlns="https://github.com/avaloniaui"
xmlns:x="http://schemas.microsoft.com/
winfx/2006/xaml"
x:Class="AvaloniaApplication5.MainWindow">
<Button Name="greetingButton">Hello World</
Button> </Window>
```

You can now access the button via an auto-generated `greetingButton` field from the code-behind class file named `MainWindow.axaml.cs`, as shown in the following code:

```
using Avalonia.Controls; namespace
AvaloniaApplication1.Views { public partial
class MainWindow : Window { public
MainWindow() { InitializeComponent(); //
Changing button content using code.
greetingButton.Content = "Goodbye Cruel
World!"; } } }
```

Handling control events

When using the code-behind pattern, you write event handlers in the code-behind file.

Event handlers are written as methods in the code-behind file, and then referenced in the XAML using an event name attribute. For example, to add a handler for a button's `Click` event:

```
<Window xmlns="https://github.com/avaloniaui"
xmlns:x="http://schemas.microsoft.com/
winfx/2006/xaml"
x:Class="AvaloniaApplication4.MainWindow">
<Button
Click="GreetingButtonClickHandler">Hello
World</Button> </Window>
```

In the code-behind file, `MainWindow.axaml.cs`:

```
public partial class MainWindow : Window {
public MainWindow() { InitializeComponent(); }
public void GreetingButtonClickHandler(object
sender, RoutedEventArgs e) { // code here. } }
```

Note that many Avalonia event handlers pass a special argument of class `RoutedEventArgs`. This includes information about how the event has been generated and propagated.

Laying out controls

When laying out controls in Avalonia, four of the most important properties are:

- `HorizontalAlignment`
- `VerticalAlignment`

- Margin
- Padding

These properties control the positioning and spacing of elements inside a container.

HorizontalAlignment

The `HorizontalAlignment` property determines how a control is positioned horizontally within its parent container, as shown in *Table 3.2*:

Description		
Aligns the control to the left of the parent		
Centers the control horizontally within the parent		
Aligns the control to the right of the parent		
Expands the control to fill the available horizontal space.		

Table 3.2: The `HorizontalAlignment` property

For example, to center a button horizontally within its parent container, as shown in the following markup:

```
<Button Content="Click Me"
HorizontalAlignment="Center" />
```

Warning! `Stretch` is the default, but explicit `Width` and `Height` values of a child take precedence so often `Stretch` has no effect.

VerticalAlignment

The `VerticalAlignment` property determines how a control is positioned vertically within its parent container, as shown in *Table 3.3*:

Description		

Aligns the control to the top of the parent		
Centers the control vertically within the parent		
Aligns the control to the bottom of the parent		
Expands the control to fill the available vertical space.		

Table 3.3: The `VerticalAlignment` property

For example, to position a button at the bottom of its parent container, as shown in the following markup:

```
<Button Content="Click Me"
VerticalAlignment="Bottom" />
```

Warning! `Stretch` is the default, but explicit `Width` and `Height` values of a child take precedence, so often `Stretch` has no effect.

Margin

The `Margin` property controls the space *outside* a control, creating distance between the control and its surroundings. It accepts four values, as shown in Table 3.4:

Descriptor (left, Top, Right, Bottom)		
Applies a 10px margin on all sides		
10px on left and right, 20px on top and bottom		
10px left, 20px top, 30px right, 40px bottom.		

Table 3.4: The `Margin` property

For example, to add 20px of space around a button on all sides, as shown in the following markup:

```
<Button Content="Click Me" Margin="20" />
```

For example, to apply:

- 10px margin on the left
- 20px margin on the top
- 30px margin on the right
- 40px margin on the bottom, as shown in the following markup:

```
<Button Content="Click Me" Margin="10, 20, 30, 40" />
```

Padding

The `Padding` property controls the space *inside* a control, between its border and content. Like `Margin`, it accepts four values, as shown in *Table 3.5*:

Property (Left, Top, Right, Bottom)	
<code>Padding</code>	
Applies a 10px padding inside all sides	
10px padding left and right, 20px padding top and bottom	
10px left, 20px top, 30px right, 40px bottom.	

Table 3.5: The `Padding` property

For example, to add 10px of increased space inside the button, making its content less cramped, as shown in the following markup:

```
<Button Content="Click Me" Padding="10" />
```

Full example

This example demonstrates how all four properties interact within a `StackPanel`, as shown in the following markup and in *Figure 3.2*:

```

<StackPanel> <Button Content="Left Aligned"
HorizontalAlignment="Left" Margin="10"/>
<Button Content="Centered"
HorizontalAlignment="Center" Padding="20"/>
<Button Content="Stretched"
HorizontalAlignment="Stretch" Margin="10,5"
Padding="10"/> <Button Content="Bottom Right"
HorizontalAlignment="Right"
VerticalAlignment="Bottom"
Margin="10,10,10,20"/> </StackPanel> </Window>

```

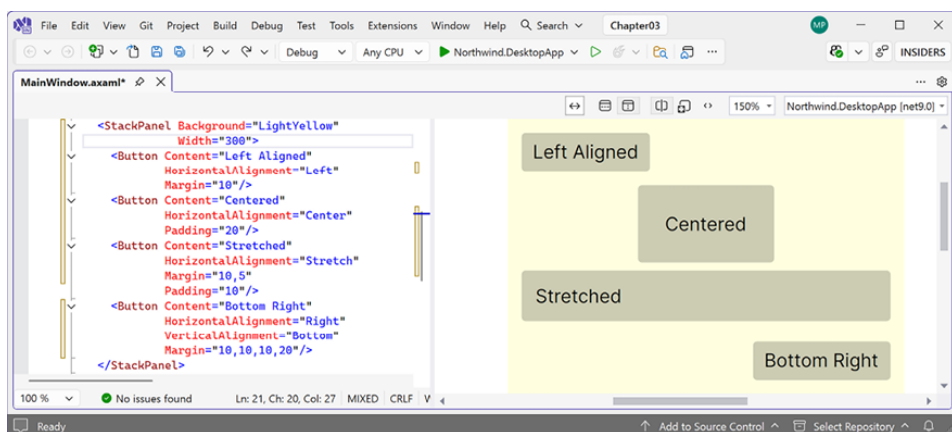


Figure 3.2: Alignments, margins, and padding

Note the following in Figure 3.2:

- The first button is left-aligned horizontally within the stack panel and has a 10-pixel margin around its outer edge.
- The second button is centered horizontally within the stack panel with no margin around its outer edge, but with a 20-pixel padding with the button between its edge and the text content.
- The third button is stretched horizontally within the stack panel and has a 10-pixel margin to its left and right, and a 5-pixel margin to its top and bottom outside edges.

- The fourth button is right-aligned horizontally within the stack panel and has a 10-pixel margin to its left, top, and right outside edges, and a 20-pixel margin to its bottom edge.

HorizontalAlignment, Margin, Padding, and VerticalAlignment combine to provide the positioning control necessary to create a complex UI. You can use the effects of each property to change child-element positioning, enabling flexibility in creating dynamic applications and user experiences.

Your key takeaways should be:

- Use HorizontalAlignment and VerticalAlignment to control positioning inside a parent.
- Use Margin to add space *outside* a control.
- Use Padding to add space *inside* a control.
- Use Stretch alignment to make elements expand inside a parent.

Panels and their attached properties for layout

Avalonia provides various panels for organizing and positioning UI elements. Each panel has specific behavior, and some panels use attached properties to control child element positioning, as shown in *Table 3.6*:

Property/Name					
StackPanel	Arranges children (vertically or horizontally) in a single line				
Grid	Arranges children in rows and columns. RowSpan, Grid.ColumnSpan				
UniformGrid	Arranges children in a grid where all cells in the grid have the same size				
WrapPanel	Aligns children to the top, bottom, left, or right of its area. The last child fills the remaining space by default				
Canvas	Allows absolute positioning of children using X/Y coordinates. Bottom				
WrapPanel	Places children in a row and wraps to the next row when space runs out.				

Table 3.6: Panels and their properties for layout

Attached properties are prefixed with the panel type, like `Grid.Row` and `DockPanel.Dock`. `StackPanel` and `WrapPanel` have normal properties named `Orientation` that do not attach to their children.

Panel elements can be nested within each other in order to produce complex layouts. In many cases, a `Grid` element can be used instead of nested panels due to its flexibility as a layout container. This can increase performance in your application by keeping unnecessary elements out of the tree.

To summarize:

- Use `Grid` for structured layouts.
- Use `StackPanel` for simple UI flows.
- Use `DockPanel` when docking elements to edges.
- Use `Canvas` for precise positioning.
- Use `WrapPanel` for flowing content.

You can see a complete list of Avalonia controls including panels at the following link: <https://docs.avaloniaui.net/docs/basics/user-interface/controls/builtin-controls>.

Styling controls

In Avalonia, styles allow you to define reusable appearance settings for UI elements. Styles are similar to CSS in web development and help maintain consistency across an application.

A style in Avalonia is a set of rules that define the appearance and behavior of UI elements. It consists of:

- **Selectors:** Determine which elements the style applies to.
- **Setters:** Define property values (for example, `Background`,

FontSize).

- **Triggers** (optional): Change properties when conditions are met.

For example, to apply a style to all buttons in the application, setting a blue background, white text, and a font size of 16, define a style as shown in the following markup:

```
<Style Selector="Button"> <Setter
Property="Background" Value="Blue"/> <Setter
Property="Foreground" Value="White"/> <Setter
Property="FontSize" Value="16"/> </Style>
```

Where styles can be defined

Styles can be defined in different places:

- Inline (directly to a control), which is simple, but not reusable, as shown in the following markup:

```
<Button Content="Click Me"
Background="Blue" Foreground="White"/>
```

- Inside a `Resources` element, which is reusable within the same scope, for example, all buttons within a window, as shown in the following markup:

```
<Window xmlns="https://github.com/
avaloniaui" Title="Style Example">
<Window.Resources> <Style
Selector="Button"> <Setter
Property="Background" Value="Blue"/>
<Setter Property="Foreground"
Value="White"/> </Style> </
```

```
Window.Resources> <StackPanel> <Button
Content="First Button"/> <Button
Content="Second Button"/> </StackPanel> </
Window>
```

- In a separate .axaml file which allows easier sharing within and between projects.

Style selectors

Selectors define which elements the style applies to, as shown in *Table 3.7*:

Selection				
Applies the style to a specific control type				
Applies the style to all elements with a specific class				
Applies the style to a specific named element				
Targets children of a specific parent				
Applies style when an element is in a specific state.				

Table 3.7: Style selectors

Here’s an example of class-based selectors, as shown in the following markup:

```
<Style Selector="Button.custom-button">
<Setter Property="Background" Value="Green"/>
<Setter Property="FontSize" Value="18"/> </
Style>
```

Now you can apply the style based on its class, as shown in the following markup:

```
<Button Classes="custom-button"
Content="Styled Button"/> <Button
Content="Unstyled Button"/>
```


This allows you to target specific elements and avoid accidental global changes that you don't intend.

Also, like CSS, an Avalonia style can be applied to a specific element using #name, as shown in the following markup:

```
<Style Selector="Button#submitButton"> <Setter  
Property="Background" Value="Red"/> </Style>
```

You can apply this to a specific button, as shown in the following markup:

```
<Button Content="Submit" Name="submitButton"/>
```

This enables precise targeting but is not reusable.

Style inheritance

You can base a style on another using BasedOn, as shown in the following markup:

```
<Style Selector="Button.base-style"> <Setter  
Property="FontSize" Value="16"/> <Setter  
Property="Padding" Value="10"/> </Style>  
<Style Selector="Button.warning"  
BasedOn="Button.base-style"> <Setter  
Property="Background" Value="Red"/> </Style>
```

Then you can apply the style, as shown in the following markup:

```
<Button Classes="base-style" Content="Normal  
Button"/> <Button Classes="warning"
```

```
Content="Warning Button"/>
```

This avoids duplicate code and encourages consistency.

Style triggers

Triggers aka pseudo-classes change properties dynamically based on conditions. This enhances the user experience. For example, you can change the background when a mouse hovers over the control, as shown in the following markup:

```
<Style Selector="Button"> <Setter
Property="Background" Value="Blue"/> <Setter
Property="Foreground" Value="White"/> <Style
Selector="Button: hover"> <Setter
Property="Background" Value="DarkBlue"/> </
Style> </Style>
```

A complete list of pseudo-classes that can be used as style triggers are shown in *Table 3.8*:

WhenActive		
Pointer (mouse, stylus, touch if relevant) is over the control bounds (hover state) <i>Button: hover equivalent</i>		
The button is being pressed / pointer down (or keyboard “pressed” equivalent)		
Control is <code>Enabled = false</code> (cannot be interacted)		
Control has input focus (keyboard or programmatic)		
Control has focus and should show the visible focus indication (often when focus comes from keyboard vs mouse)		
Either the control itself has focus or some descendant has focus. Applied to container controls (and possibly relevant in templated parts), but less relevant directly to simple button content		
For toggleable or checkable controls like <code>ToggleButton</code> ,		

	RadioButton, CheckBox, when their “checked” state is true. Not for a plain Button		
	For controls supporting three-states like a CheckBox with IsThreeState and its state is neither checked nor unchecked with a null value.		

Table 3.8: Pseudo-classes aka style triggers

You can learn more about pseudo-classes at the following link: <https://docs.avaloniaui.net/docs/reference/styles/pseudo-classes>

Global styles

To apply styles across the entire app, define them in `App.axaml`, as shown in the following markup:

```
<Application.Styles> <Style Selector="Button">
<Setter Property="Background" Value="Blue"/>
</Style> </Application.Styles>
```

This ensures uniform design and avoids duplication.

You can learn more about styling at the following link: <https://docs.avaloniaui.net/docs/basics/user-interface/styling/>.

You’ve seen how Avalonia implements XAML for building its user interfaces using controls, panels, and styles, and that it is very similar to the XAML used by .NET MAUI (and WPF).

Enough theory! Let’s build a desktop app using Avalonia.

Creating an Avalonia project

Let's start by creating a new Avalonia MVVM project:

1. In your preferred code editor, add a new project, as defined in the following list:

- Project template: **Avalonia .NET MVVM App (AvaloniaUI)** / `avalonia.mvvm`
- Project file and folder: `Northwind.DesktopApp`
- Solution file and folder: `ModernApps`
- **MVVM Toolkit**: `CommunityToolkit`
- **Use Compiled Bindings**: `Selected`
- **Remove View Locator**: `Cleared`

8. In the `Northwind.DesktopApp.csproj` project file, remove any package reference version attributes (because we are using CPM), as shown in the following markup:

```
<ItemGroup> <PackageReference
Include="Avalonia" /> <PackageReference
Include="Avalonia.Desktop" />
<PackageReference
Include="Avalonia.Themes.Fluent" />
<PackageReference
Include="Avalonia.Fonts.Inter" /> <!--
Condition below is needed to remove
Avalonia.Diagnostics package from build
output in Release configuration.-->
<PackageReference
Include="Avalonia.Diagnostics">
<IncludeAssets
Condition="'$(Configuration)' !=
'Debug'">None</IncludeAssets>
<PrivateAssets
Condition="'$(Configuration)' !=
```

```
'Debug'">All</PrivateAssets> </
PackageReference> <PackageReference
Include="CommunityToolkit.Mvvm" /> </
ItemGroup>
```

9. In the ViewModels folder, in MainWindowViewModel.cs, change the Greeting property to say, “Welcome to Apps and Services with .NET 10 with Avalonia!”, as shown in the following code:

```
public string Greeting { get; } = "Welcome
to Apps and Services with .NET 10 with
Avalonia!";
```

10. In MainWindow.axaml, change the Title property to Northwind Desk App, as shown highlighted in the following markup:

```
Title="Northwind Desktop App">
```

11. Start the Northwind.DesktopApp project without debugging, and note the appearance of the default project template app on Windows, as shown in *Figure 3.3*:

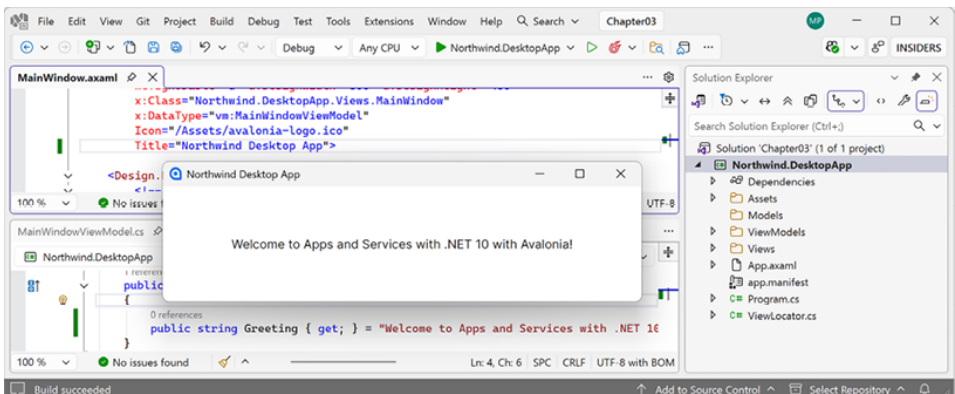


Figure 3.3: A simple Avalonia app on Windows

1. Close the app.

Next, we will review how to layout desktop apps using the `DockPanel` layout control.

DockPanel for layout in desktop apps

Mobile apps often use layout controls like `StackPanel` in a vertical layout because mobile devices are tall and skinny. Desktop apps often use layout controls like `DockPanel` because they need to support a larger, more variable layout and make the best use of space.

`DockPanel` is a layout control in Avalonia that arranges its child elements along its edges. You can dock elements to the top, bottom, left, or right of the panel, and one child can occupy the remaining space. Some important properties of `DockPanel` are shown in *Table 3.9*:

Description		
<code>DefaultChildFill</code> If true, the last child control automatically fills the remaining space		
<code>AttachedProperty</code> (meaning any controls within the <code>DockPanel</code> are given this property) that determines which edge of the panel the child element docks to. Values are: <code>Top</code> , <code>Bottom</code> , <code>Left</code> , <code>Right</code> .		

Table 3.9: Important `DockPanel` properties

Let’s see a simple example of using `DockPanel`:

1. In the `Northwind.DesktopApp` project, in the `Views` folder, add a new **Avalonia Windows (AvaloniaUI)** file named `DockPanelWindow.axaml`.
2. Replace the **Welcome to Avalonia!** text with a `DockPanel` containing five elements, as shown in the following markup:

```

<DockPanel LastChildFill="True">
<TextBlock DockPanel.Dock="Top"
Background="LightBlue" Text="Top Docked" /
> <TextBlock DockPanel.Dock="Bottom"
Background="LightGreen" Text="Bottom
Docked" /> <TextBlock
DockPanel.Dock="Left"
Background="LightCoral" Text="Left Docked"
/> <TextBlock DockPanel.Dock="Right"
Background="LightGoldenrodYellow"
Text="Right Docked" /> <TextBlock
Background="LightGray" Text="Main Content
Area (Last Child Fill)" /> </DockPanel>

```

To explain the preceding markup:

- Children like the `TextBlock` elements are positioned in the order they appear in their parent.
- The top and bottom areas are occupied by `TextBlocks` with `DockPanel.Dock="Top"` and `DockPanel.Dock="Bottom"`.
- The left and right areas are filled by `TextBlocks` with `DockPanel.Dock="Left"` and `DockPanel.Dock="Right"`.
- The center area is occupied by the last child (`TextBlock` with gray background) because `LastChildFill="True"` (default is `true`).

1. Build the project.
2. Note the designer shows the layout, with four colored elements around the edges, and the light gray element taking up the remaining space in the middle, as shown in *Figure 3.4*:

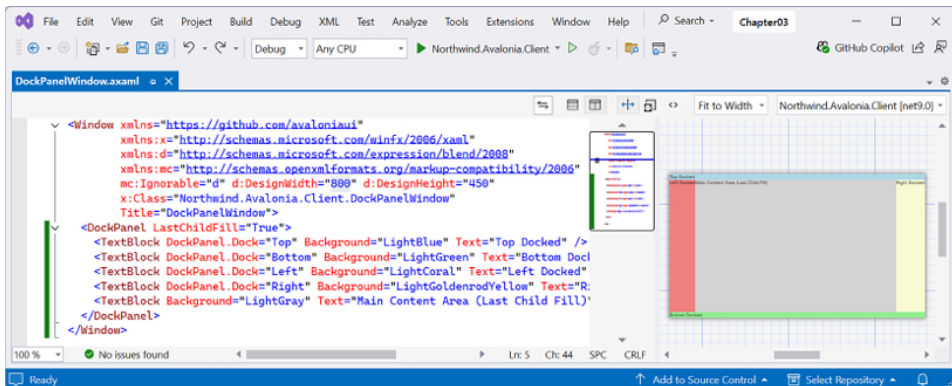


Figure 3.4: DockPanel example layout with five elements

1. In the Views folder, in `MainWindow.axaml`, change the `TextBlock` element to a `Button` element, and set its `Content` property instead of its `Text` property, as shown in the following markup:

```
<Button Content="{Binding Greeting}"
        HorizontalAlignment="Center"
        VerticalAlignment="Center" />
```

2. In the `Button` element, set its `Click` event, and select **New Event Handler**, as shown in Figure 3.5:

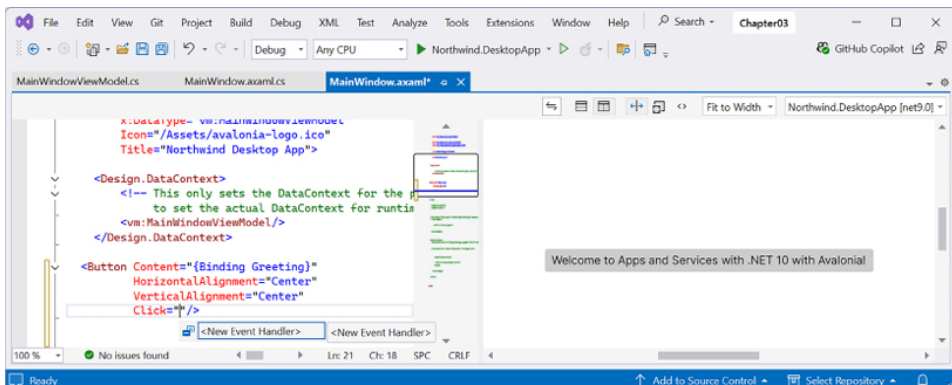


Figure 3.5: Setting an event handler in XAML

1. Note that it sets the `Click` event handler to `Button_Click`.
2. In `MainWindow.axaml.cs`, note the event handler written in C#, and add statements to create a new instance of the `DockPanelWindow` and show it, as shown in the following code:

```
private void Button_Click(object? sender,
    Avalonia.Interactivity.RoutedEventArgs e)
{ DockPanelWindow dpw = new(); dpw.Show();
}
```

3. Start the `Northwind.DesktopApp` project without debugging.
4. Click the button and note that the window with a `DockPanel` appears.
5. Resize the window and note the panels stay docked to their side, as shown in *Figure 3.6*:

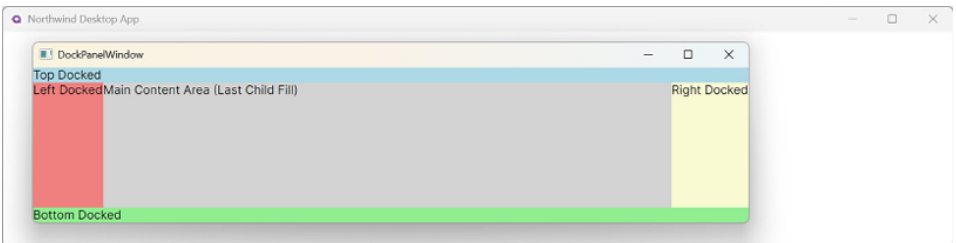


Figure 3.6: Resizing a window that uses a DockPanel for layout

1. Close the app.

Good practice for DockPanel

Use `DockPanel` when:

- You need a structured layout with fixed header, footer, sidebars, and a flexible center.
- Your layout requires dynamic resizing with sections expanding or

shrinking automatically.

- It's best for sidebars, toolbars, and filling remaining space dynamically.

Avoid `DockPanel` if:

- You need precise positioning (use `Grid` or `Canvas` instead).
- You want equal-width columns or rows (use `UniformGrid` or `Grid`).
- A simple stack layout is enough (use `StackPanel`).

Some controls like `UniformGrid` are not in the core controls. To use them you may need to reference additional packages. For example, `UniformGrid` is in the `Avalonia.Controls.Panels` package. After referencing this package, import the namespace in any `.axaml` file:

```
xmlns:panels="clr-namespace:Avalonia.Controls;assembly=
```

Now let's change the main window to show a list of categories with their products.

Implementing a simple desktop app for data

We will reference the EF Core model that you created in [Chapter 1](#), *Introducing Apps and Services with .NET*:

1. In the `Northwind.Blazor.csproj` project file, treat warnings as errors, and add a project reference to the Northwind database context project, as shown in the following markup:

```
<ItemGroup> <ProjectReference Include="..\
\Northwind.DataContext
\Northwind.DataContext.csproj" /> </
ItemGroup>
```

The Include path must not have a line break.

1. Build the Northwind.Blazor project.
2. In the ViewModels folder, add a new class file named CategoryViewModel.cs.
3. In CategoryViewModel.cs, define a class that derives from the Category entity model, as shown in the following code:

```
using Northwind.EntityModels; // To use
Category entity model. namespace
Northwind.DesktopApp.ViewModels; public
class CategoryViewModel : Category {
public string? PicturePath { get =>
$"avares://{Northwind.DesktopApp/Assets/
category{CategoryId}-small.jpeg"; } }
```

4. In MainWindowViewModel.cs, add statements to use NorthwindContext to load data from the JSON file into a view model that has properties to store categories and products for data binding, as shown highlighted in the following code:

```
using Northwind.EntityModels; // To use
Category and Product. using
System.Collections.ObjectModel; // To use
ObservableCollection<T>. using
System.Linq; // To use LINQ methods.
```

```

namespace Northwind.DesktopApp.ViewModels
{
    public partial class MainWindowViewModel
        : ViewModelBase {
        public string Greeting {
            get; } = "Welcome to Apps and Services
            with .NET 10 with Avalonia!";
        public
        ObservableCollection<CategoryViewModel>
        Categories {
            get; } = [];
        public
        ObservableCollection<Product> Products {
            get; } = [];
        private Category?
        _selectedCategory;
        public Category?
        SelectedCategory {
            get =>
            _selectedCategory;
            set {
                _selectedCategory
                = value;
                Products.Clear();
                if (value !=
                null) {
                    foreach (var product in
                    value.Products)
                        Products.Add(product);
                }
            }
        }
        public MainWindowViewModel() {
            using
            NorthwindContext db = new();
            CategoryViewModel[]? categories =
            db.Categories .Select(c => new
            CategoryViewModel {
                CategoryId =
                c.CategoryId,
                CategoryName =
                c.CategoryName,
                Description =
                c.Description,
                Picture = c.Picture,
                Products = c.Products
            }).ToArray();
            if
            (categories is null)
                return;
            foreach
            (CategoryViewModel category in
            categories)
            {
                Categories.Add(category);
            }
            SelectedCategory = Categories[0];
        }
    }
}

```

5. In the Views folder, in `MainWindows.axaml`, comment out the Button, and then add statements to render a `DockPanel` with lists of categories and products, as shown in the following markup:

```

<!--<Button Content="{Binding Greeting}"

```

```

HorizontalAlignment="Center"
VerticalAlignment="Center"
Click="Button_Click"/>--> <DockPanel>
<ListBox DockPanel.Dock="Left" Width="200"
ItemsSource="{Binding Categories}"
SelectedItem="{Binding SelectedCategory}"
Background="Gainsboro">
<ListBox.ItemTemplate> <DataTemplate>
<TextBlock Text="{Binding CategoryName}"/>
</DataTemplate> </ListBox.ItemTemplate> </
ListBox> <TextBlock DockPanel.Dock="Top"
Text="{Binding
SelectedCategory.CategoryName}"
FontSize="20" FontWeight="Bold"
Margin="10"/> <TextBlock
DockPanel.Dock="Top" Text="{Binding
SelectedCategory.Description}"
FontSize="18" Margin="10,0,10,10" />
<ListBox ItemsSource="{Binding Products}"
VerticalAlignment="Stretch">
<ListBox.ItemTemplate> <DataTemplate>
<StackPanel Orientation="Horizontal">
<TextBlock Text="{Binding ProductId}"
Width="50"/> <TextBlock Text="{Binding
ProductName}" Width="200"/> <TextBlock
Text="{Binding UnitPrice,
StringFormat='C2', ConverterCulture='en-
US'}" Width="100" TextAlignment="Right"/>
<TextBlock Text="{Binding UnitsInStock}"
Width="75" TextAlignment="Right"/> </
StackPanel> </DataTemplate> </
ListBox.ItemTemplate> </ListBox> </
DockPanel>

```

6. Build the project, and note that the designer shows the layout including

the data, as shown in *Figure 3.7*:

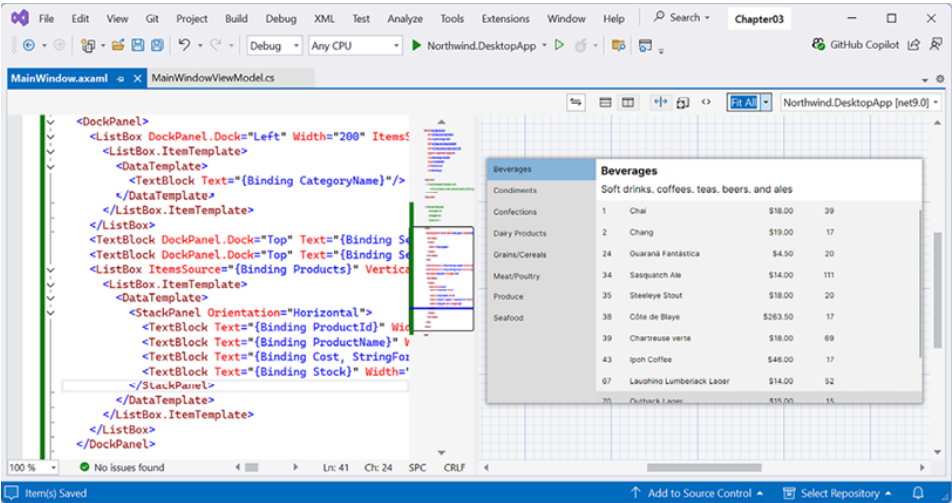


Figure 3.7: Avalonia designer shows the layout with data

1. Start the project without debugging.
2. Note the desktop app resizes correctly, making the most of the available space, as shown in *Figure 3.8*:

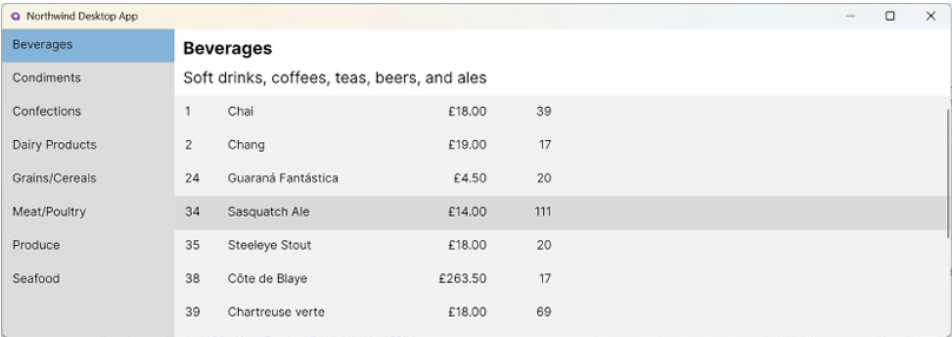


Figure 3.8: Northwind desktop app with loaded data

1. Close the app.

Adding a typical menu

Unlike mobile apps that might use a toolbar with three or four icons to access

actions to perform, or different parts of the user interface, desktop apps are more complex with more commands, so they tend to use menus.

A menu in Avalonia is defined by a `<Menu>` instance that should be contained within a `DockPanel` and uses the `attached` property to pin it to the top edge of the window. Menus use `<MenuItem Header="...">` and `<Separator>` elements to define the nested structure.

To activate a menu item using the keyboard, the user can press `Alt + <letter>`, which is called an accelerator. To indicate the letter that can be pressed, prefix it with an underscore. That letter will appear underscored in the user interface. For example, to press `Alt + F` to activate a **F**ile menu, set its `Header` to `"_File"`.

To define an alternative keyboard shortcut, like `Ctrl + C` for Copy, set the `InputGesture` property.

To add event handlers, set the `Click` event, like you would with a `Button`, as shown in the following markup:

```
<MenuItem Header="_Open" Click="OnOpenClicked"
InputGesture="Ctrl+O" />
```

To bind to an MVVM command, set the `Command` property, as shown in the following markup:

```
<MenuItem Header="_Open" Command="{Binding
OpenCommand}" InputGesture="Ctrl+O" />
```

Let's add a typical menu to our desktop app:

1. In `MainWindow.axaml`, add statements to define a typical menu docked to the top of the `DockPanel`, as shown in the following markup:

```

<DockPanel> <Menu DockPanel.Dock="Top">
  <MenuItem Header="_File"> <MenuItem
Header="_New" InputGesture="Ctrl+N" />
  <MenuItem Header="_Open"
InputGesture="Ctrl+O" /> <MenuItem
Header="_Save" InputGesture="Ctrl+S" />
  <Separator/> <MenuItem Header="E_xit"
Click="ExitMenu_Click" InputGesture="Alt
+F4" /> </MenuItem> <MenuItem
Header="_Edit"> <MenuItem Header="_Undo"
InputGesture="Ctrl+Z" /> <MenuItem
Header="_Redo"/> <Separator/> <MenuItem
Header="Cu_t" InputGesture="Ctrl+X" />
  <MenuItem Header="_Copy"
InputGesture="Ctrl+C" /> <MenuItem
Header="_Paste" InputGesture="Ctrl+V" />
</MenuItem> <MenuItem Header="_Help">
  <MenuItem Header="_About" /> </MenuItem>
</Menu> ...

```

2. In `MainWindow.axaml.cs`, add statements to define the event handler, as shown in the following markup:

```

private void ExitMenu_Click(object?
sender,
Avalonia.Interactivity.RoutedEventArgs e)
{ Close(); }

```

3. Start the project.
4. Use the **File | Exit** menu to close the app, as shown in *Figure 3.9*:

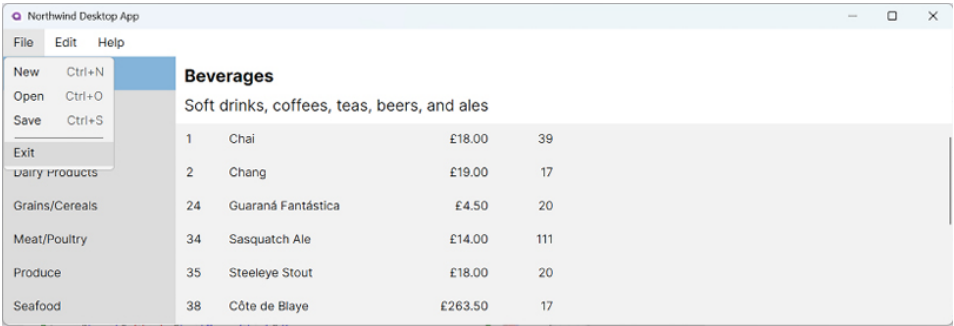


Figure 3.9: File menu with accelerator keys in a desktop app

You’ve now built a simple type of desktop app that binds hierarchical data like categories and products to a user interface and added a menu to provide quick yet comprehensive access to the app’s functionality. Now let’s add media assets like images.

Referencing assets like images

In Avalonia, assets refer to external resources such as images, icons, fonts, and data files used within an application. Avalonia provides a structured way to manage and reference these assets.

Avalonia assets are typically stored inside the project’s Assets folder or another directory marked as an asset directory in the .csproj file. Assets should be placed in a directory such as Assets/ or Resources/. By default, Avalonia expects assets to be inside the project structure.

Adding assets to a project

Ensure that assets are properly included in the project by making sure that this is in your .csproj file, as shown in the following configuration:

```
<ItemGroup> <AvaloniaResource Include="Assets
\\**" /> </ItemGroup>
```

This tells Avalonia to package everything inside `Assets/` as part of the application resources including subfolders.

Referencing assets in a project

Avalonia uses the `avares://` scheme to reference assets. The general format is shown in the following example:

```
avares://<AssemblyName>/Assets/<FileName>
```

For example, to use an image, as shown in the following markup:

```
<Image Source="avares://MyAvaloniaApp/Assets/
Images/logo.png"/>
```

To use images dynamically, load them using `Bitmap`, as shown in the following code:

```
using Avalonia.Media.Imaging; // To use
Bitmap. using Avalonia.Platform; // To use
AssetLoader. Uri uri = new("avares://
MyAvaloniaApp/Assets/Images/logo.png"); Bitmap
bitmap = new(AssetLoader.Open(uri));
imageControl.Source = bitmap;
```

The `IAssetLoader` service allows retrieving assets dynamically, as shown in the following code:

```
using Avalonia.Platform; // To use
IAssetLoader. using System.IO; // To use
StreamReader. Uri uri = new("avares://
```

```
MyAvaloniaApp/Assets/Data/config.json");
Stream stream =
AvaloniaLocator.Current.GetService<IAssetLoader>().Open(
using StreamReader reader = new(stream);
string content = reader.ReadToEnd();
```

Note the following about the preceding code:

- We import some namespaces:
 - `Avalonia.Platform` gives access to Avalonia's platform abstractions. `IAssetLoader` is Avalonia's service for loading embedded resources (things like images, XAML snippets, JSON configs that you bundle with your app).
 - `System.IO` provides `StreamReader`, which lets you read the text contents of a `Stream`.
- It creates a `Uri` object pointing to a resource inside the Avalonia app:
 - The `avares://` scheme is Avalonia's special URI format for addressing resources embedded in the application. The syntax is: `avares://<AssemblyName>/<Path/Within/Project>`. So, `MyAvaloniaApp` is the assembly name, and `Assets/Data/config.json` is the relative path to the file inside the project's resources.
 - To make this work, the `config.json` file must be set to **Build Action: AvaloniaResource** in your project.
- It loads a raw data stream for the embedded `config.json`:
 - `AvaloniaLocator.Current` is Avalonia's service locator (a kind of global DI container).
 - `GetService<IAssetLoader>()` retrieves the app's current asset loader.
 - `.Open(uri)` tells the loader to open the resource

specified by the `avares://` URI, returning it as a `Stream`.

- It wraps the `Stream` in a `StreamReader` for easy text reading:
 - `ReadToEnd()` reads the entire file as a `string`.
 - The `using` declaration ensures the `StreamReader` (and underlying stream) is disposed properly when the scope ends.
- `content` now contains the text of your `config.json` file, loaded from your embedded Avalonia resource bundle. You could then parse it with `System.Text.Json` or `Newtonsoft.Json` to configure your app.

Adding images to the project

Let's add an image to the desktop app:

1. In the `Assets` folder, add images for each of the eight categories and an image for all of them. You can download these images from the following link: <https://github.com/markjprice/apps-services-net10/tree/main/code/ModernApps/Northwind.DesktopApp/Assets>.
2. In `MainWindow.axaml`, under the menu, add statements to wrap the list box that shows categories in a stack panel docked to the left, and add an image of categories above the list box, as shown highlighted in the following markup:

```
<StackPanel DockPanel.Dock="Left"> <Image  
Source="/Assets/categories-small.jpeg"  
Height="135" Width="200" /> <ListBox  
Width="200" ItemsSource="{Binding  
Categories}" SelectedItem="{Binding  
SelectedCategory}" Background="Gainsboro">  
<ListBox.ItemTemplate> <DataTemplate>
```

```
<TextBlock Text="{Binding CategoryName}"/>
</DataTemplate> </ListBox.ItemTemplate> </
ListBox> </StackPanel>
```

3. Start the app and note the image in the top-left corner of the desktop app, as shown in *Figure 3.10*:

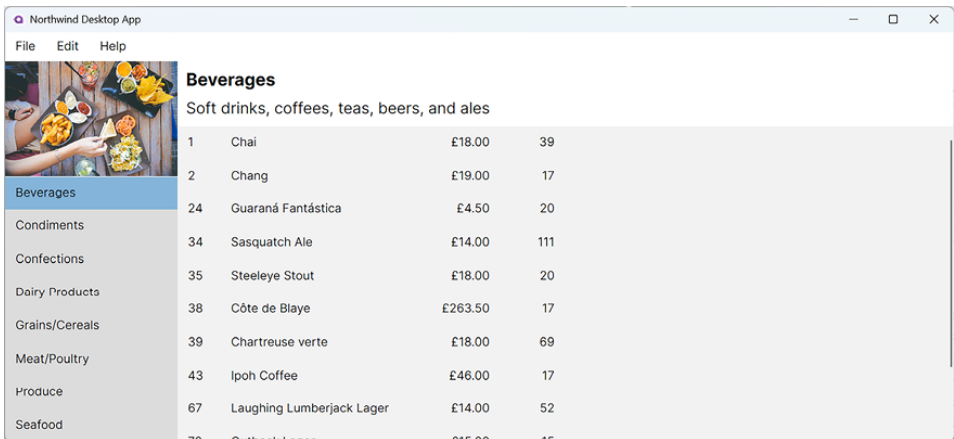


Figure 3.10: An image of categories

Practicing and exploring

Test your knowledge and understanding by answering some questions, getting some hands-on practice, and exploring this chapter's topics with deeper research.

Exercise 3.1 – Online material

Avalonia has a complete sample app, Music Store App, as shown in *Figure 3.11*, with video tutorial at the following link: <https://docs.avaloniaui.net/docs/tutorials/music-store-app/>

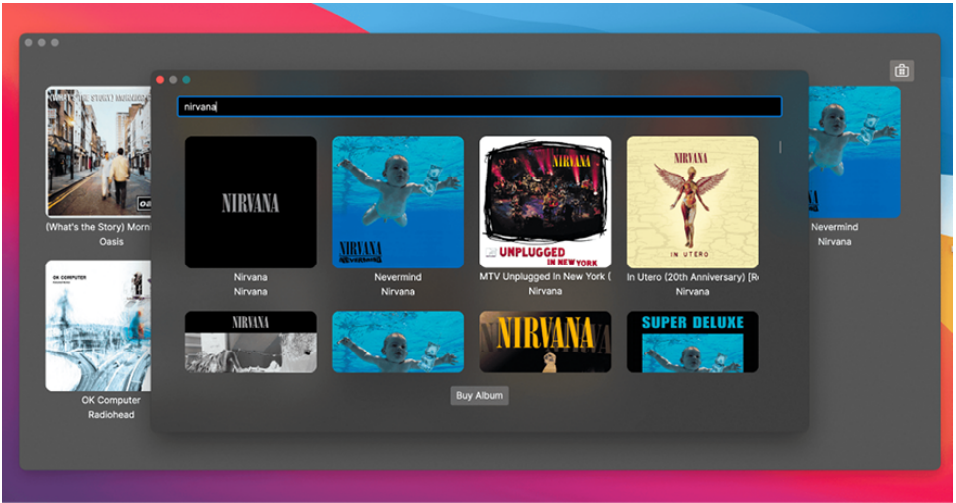


Figure 3.11: Music Store App user interface

If you have experience with WPF, then you can learn how to migrate to Avalonia at the following link: <https://docs.avaloniaui.net/docs/get-started/wpf/>

Exercise 3.2 – Practice exercises

Create a GroupBox Using HeaderedContentControl

Avalonia doesn't have a built-in `GroupBox` control, but you can replicate its functionality and appearance using a `HeaderedContentControl` with custom styling. This control includes a header section and a content area, making it ideal for grouping related UI elements.

If you get stuck, then review the code solution in the GitHub repository at the following link: <https://github.com/markjprice/apps-services-net10/blob/main/code/ModernApps/Northwind.DesktopApp/Views/GroupBox.axaml>

Tutorial samples

Review sample Avalonia projects at the following link: <https://docs.avaloniaui.net/docs/tutorials/samples>

Exercise 3.3 – Test your knowledge

Answer the following questions:

1. You create a new Avalonia app with the project template, and it runs fine using the example code:

```
<Window xmlns="https://github.com/avaloniaui" ...> Welcome to Avalonia! </Window>
```

But when you add a button under the existing text inside the `<Window>` element, as shown highlighted in the following code, you get a compile error: Unable to find a setter that allows multiple assignments to the property Content of type Avalonia.Controls.Avalonia.Controls.ContentControl. What is the problem, and how do you fix it?

```
<Window xmlns="https://github.com/avaloniaui" ...> Welcome to Avalonia! <Button>Test</Button> </Window>
```

1. Why does Avalonia use the file extension `.axaml`?
2. What are attached properties, and how are they used?
3. How does Avalonia handle the layout and positioning of controls?
4. How do you define styles for controls?

5. What is the purpose of data templates?
6. How can you bind a property in a ViewModel to a control?
7. How does Avalonia's rendering pipeline differ from WPF's?
8. What is the purpose of `INotifyPropertyChanged` and `ReactiveObject` in Avalonia MVVM applications?
9. How do you implement theming and dynamic styling?

Appendix A, Answers to the Test Your Knowledge Questions, is available to download from the following link:

https://github.com/markjprice/apps-services-net10/blob/main/docs/B31467_Appendix%20A.pdf.

Exercise 3.4 – Explore topics

Use the links on the following page to learn more details about the topics covered in this chapter:

<https://github.com/markjprice/apps-services-net10/blob/main/docs/book-links.md#chapter-3---building-desktop-apps-using-avalonia>

Summary

In this chapter, you learned:

- About the differences between .NET MAUI and Avalonia and why Avalonia is better for desktop app development.
- How to work with controls, including laying out the user interface and styling controls.
- How to use `DockPanel` to provide a flexible layout for a desktop app.
- How to manage assets like images.

In the next chapter, you will learn how to build web apps using ASP.NET Core Blazor.

Get This Book

UNLOCK NOW

ion and Exclusive

Scan the QR code
book by name, co



. Search for this
ps on the page.

***Note:** Keep your invoice handy. Purchases made directly from Packt don't require one.*

4

Building Web Apps Using Blazor

The goal of this chapter is to learn how to build web apps using Blazor. These can be rich and interactive user interfaces that render as HTML and CSS to provide cross-platform browser support.

We will start by learning about Blazor concepts, its hosting models and components, and how to perform common tasks in a web app, such as routing requests to page components, passing route parameters, and managing navigation and shared layouts.

Then we will learn how to build custom components, including both server-side rendered and interactive components, such as progress bars, dialog boxes, and alert components. Finally, we will build a data component that works with entities loaded from a file.

This chapter will cover the following topics:

- Understanding Blazor
- Building Blazor components
- Building a Blazor data component

Understanding Blazor

Blazor is Microsoft's framework for web component development built on

.NET.

There are many advantages to using .NET for client-side web development. You can write 99% of your code using C# instead of JavaScript and interop with JavaScript modules for the other 1%. You can share business logic between the server and the client. Blazor implements .NET Standard as well as the latest .NET 8 libraries, so you can use the extensive older .NET libraries, both from Microsoft and third parties.

In the previous edition of this book, this chapter covered **Blazor WebAssembly**, a hosting model where the entire Blazor app and the .NET runtime were downloaded to the browser and executed there. One of the problems with Blazor WebAssembly is a slow initial startup experience for the visitor because a lot needs to be downloaded and executed on the client.

Many .NET developers were frustrated with having to choose between different technologies to build web apps, because none of them are perfect and all have pros and cons.

In this edition of the book, this chapter covers the unified Blazor Full Stack model introduced with .NET 8. This enables you to mix the best of all worlds in a single project, including the following:

- Blazor components that execute on the client-side using WebAssembly. This replaces what is possible with a standalone Blazor WebAssembly project.
- Blazor components that execute on the server-side and communicate live with the **Document Object Model (DOM)** in the browser, using SignalR to perform updates. This replaces what is possible with a Blazor Server project.
- Blazor components that provide **static server rendering (SSR)** and return an HTTP response, with static content that does not interact live with the server. This replaces what is possible with Razor Pages or Razor Views used in traditional ASP.NET Core websites.
- Blazor components that provide server-side streaming so that some

content is shown to the visitor as soon as possible, while the rest streams to the browser in the background. This is a brand-new feature.

- A future release with .NET 11 or 12 will enable Blazor to execute in any .NET process, like a console app, so that it can be used as a **static site generator** (SSG).

Blazor hosting models

Blazor has multiple hosting models to choose from:

- **Blazor Server:** All the components execute on the web server, and user interface updates are sent to the browser using SignalR. The nature of Blazor Server provides some key benefits, including complete .NET API support, direct access to all server-side resources like databases, fast initial load time, and your code is protected because it never leaves the server. This hosting model was introduced with .NET Core 3.0 in November 2019.
- **Blazor WebAssembly:** All the components execute in the web browser like other **single page application** (SPA) frameworks, for example, React and Angular. Your .NET assemblies and the .NET runtime are downloaded to the browser and cached for future use. The nature of Blazor WebAssembly provides some key benefits, including the ability to run the app offline when not connected to the network, to host the app on a static website or serve it from a **content delivery network** (CDN), and to offload processing to the client, which increases scalability. This hosting model was introduced as an extension to .NET Core 3.1 in May 2020 and was built-in with .NET 5 in November 2020.
- **Blazor Hybrid/.NET MAUI Blazor App:** All the components execute in a local web view hosted in a native client app. The app can be built using .NET MAUI if the app needs to be cross-platform, or using Windows Presentation Foundation or Windows Forms if you are only targeting Windows. The main benefit of Blazor Hybrid compared to the first two hosting models is access to native client capabilities that can

provide a better user experience. This hosting model was introduced with .NET 7 in November 2022.

- **Blazor Full Stack:** Components can execute on the server and generate static markup, but each individual component can be switched to any of the following: streaming rendering, interactive server-side with live updates of the DOM using SignalR, or interactive client-side with WebAssembly. This latest hosting model was formerly known as Blazor United during .NET 8 previews. It was introduced as Blazor Full Stack with .NET 8 in November 2023. In .NET 10, it is simply known as Blazor.

Good practice: For new projects, **Blazor Web App** should be your choice of project template. If you need a pure SPA project that can be hosted on Azure Static Web Apps or a CDN, then **Blazor WebAssembly Standalone App** will be your best choice because Blazor Web App requires a web server. For static websites, Blazor WebAssembly is still the right solution rather than Blazor Full Stack.

Instead of multiple *hosting models*, Blazor Full Stack has multiple equivalent *rendering modes*. The Blazor Server project template that hosted and executed its code on the server-side has been replaced by the *interactive server rendering mode*. The Blazor WebAssembly project templates that could be hosted even on a static website and execute their code on the client-side can now be replaced by the *interactive WebAssembly rendering mode*.

Blazor supports the latest version of all four major web browsers: Chrome, Firefox, Edge, and Safari, on mobile and desktop platforms. Blazor Hybrid supports the latest web view components on the three major platforms: Chrome on Android, Safari on iOS and macOS, and Edge WebView2 on Windows.

The official Blazor documentation has a useful table

to help you choose between the hosting models.

You can find it at the following link: <https://learn.microsoft.com/en-us/aspnet/core/blazor/hosting-models#which-blazor-hosting-model-should-i-choose>.

Blazor components and routing

Blazor is all about **components**. A component is a part of a web app, like a button, a grid, a form for gathering input from the visitor, or even a whole page. Components can be reused and nested to build more complex components.

A Blazor component usually consists of a Razor file with the file extension `.razor`. Like Razor views in ASP.NET Core MVC or Razor Pages, Razor files used by Blazor components easily mix HTML and C# code. As well as the HTML elements that make up the user interface parts, and the CSS used to style them, the Razor file also has a code block to implement event handling, properties, and other statements to provide the functionality of the component.

For example, a Blazor component named `ProgressBar.razor` could implement a progress bar using Bootstrap. It might define parameters for a minimum, maximum, and the current value of the progress bar, and have Boolean parameters to enable animation style and show the current value as text, as shown in the following markup:

```
<div class="progress"> <div class="progress-  
bar progress-bar-striped bg-info @(IsAnimated  
? " progress-bar-animated" : "")"  
role="progressbar" aria-label="@LabelText"  
style="width: @Value%" aria-valuenow="@Value"  
aria-valuemin="@Minimum" aria-  
valuemax="@Maximum"> @(ShowValue ? Value + "%" : "") </div> </div> @code { [Parameter] public
```

```

int Value { get; set; } = 0; [Parameter]
public int Minimum { get; set; } = 0;
[Parameter] public int Maximum { get; set; } =
100; [Parameter] public bool IsAnimated { get;
set; } = false; [Parameter] public bool
ShowValue { get; set; } = false; [Parameter]
public string? LabelText { get; set; } =
"Progress bar"; }

```

Note the following about the preceding code:

- The outer `<div class="progress">` is the container.
- The inner `<div class="progress-bar ..." ...>` is the actual bar whose width visually represents progress. `progress-bar-striped` adds a striped pattern. `bg-info` gives it the Bootstrap “info” blue color.
- `@(IsAnimated ? " progress-bar-animated" : "")`: Blazor evaluates this inline C#. If `IsAnimated` is `true`, it adds the `progress-bar-animated` class (which makes the stripes move). If not, nothing is added.
- `role="progressbar"` and `aria-*` attributes are accessibility attributes (screen readers, ARIA).
- `style="width: @Value%"` dynamically sets how full the bar looks, based on the `Value` and controls how much of the bar is filled.
- `@(ShowValue ? Value + "%" : "")`: Inline conditional. If `ShowValue` is `true`, display 25% (or whatever `Value` is). If `false`, display nothing inside the bar.
- In the `@code` block, `[Parameter]` means these properties can be set by the component’s user.

To embed an instance of the component on a page, you use the component name as if it were an HTML element and set its parameters using HTML attributes, as shown in the following markup:

```
<ProgressBar Value="25" IsAnimated="true"
ShowValue="true" LabelText="Progress of
database deletion" />
```

Note the following about the preceding code:

- Renders a bar 25% full.
- The striped pattern animates because `IsAnimated="true"`.
- Displays 25% inside the bar because `ShowValue="true"`.
- For accessibility, screen readers will read it as “Progress of database deletion 25%.”

Blazor routing to page components

The `Router` component in the `App.razor` file enables routing to components, as shown in the following markup:

```
<Router
AppAssembly="@typeof(Program).Assembly">
  <Found Context="routeData"> <RouteView
RouteData="@routeData"
DefaultLayout="@typeof(Layout.MainLayout)" />
  <FocusOnNavigate RouteData="@routeData"
Selector="h1" /> </Found> </Router>
```

The `Router` component scans the assembly specified in its `AppAssembly` parameter for components decorated with the `[Route]` attribute, registering their URL paths.

If a route match is found, then the context of the request is stored in a variable named `routeData` and passed to the matching Razor file. The default layout is set to use a class defined in the file named `MainLayout.razor`.

The `FocusOnNavigate` component has a `Selector` property that must be

set to a valid CSS selector. This could be a tag selector like the default `h1`, or a more specific CSS selector that uses a CSS class or an ID. The setting is common across all components in your app, so you will need to set one that works across all your components. In the Razor file, the focus is set to the first `<h1>` element. If the Razor file contains a form, then you might want to set the first form input element, like a text box, to have the focus.

Comparing Blazor routing with MVC routing

Let's compare Blazor routing with MVC routing, traditionally the most popular ASP.NET Core technology for building websites. In a typical ASP.NET Core MVC project, an MVC controller could be decorated with the `[Route]` attribute, as shown in the following code:

```
[Route("customers")] public class  
CustomersController {
```

An HTTP GET request to the relative path `/customers` would be matched to the route.

To create an equivalent routable page component, add the `@page` directive to the top of a component's `.razor` file, as shown in the following markup:

```
@page "customers"
```

A page component can have multiple `@page` directives to register multiple routes.

If you were to write code that uses reflection to find the `component` class generated for you from the Razor markup file, then you would discover that it is decorated with the `[Route]` attribute due to the

@page directive.

At runtime, the page component is merged with any specific layout that you have specified, just like an MVC view or Razor Page would be. By default, Blazor project templates define a file named `MainLayout.razor` as the layout for page components.

Good practice: By convention, put routable page Blazor components in the `Components\Pages` folder and non-page components in the `Components` folder. The `_Imports.razor` file imports all components in this folder in your project with a statement like `@using <project_name>.Components`, for example, `@using Northwind.Blazor.Components`.

How to pass route parameters

Blazor routes can include case-insensitive named parameters, and your code can most easily access the passed values by binding the parameter to a property in the code block, using the `[Parameter]` attribute, as shown in the following markup:

```
@page "/employees/{country}" <div>Country
parameter as the value: @Country</div> @code {
[Parameter] public string Country { get; set;
} }
```

The recommended and easiest way to handle a parameter that should have a default value when it is missing is to suffix the parameter in the route with `?`, and use the `null` coalescing operator in the `OnParametersSet` method, as shown highlighted in the following markup:

```
@page "/employees/{country?}" <div>Country
parameter as the value: @Country</div> @code {
[Parameter] public string? Country { get; set;
} protected override void OnParametersSet() {
// If the automatically set property is null
// then set its value to USA. Country =
Country ?? "USA"; } }
```

Setting parameters from a query string

You can also set component properties using parameters from a query string, as shown in the following code:

```
[Parameter] [SupplyParameterFromQuery(Name =
"country")] public string? Country { get; set;
}
```

The benefits include deep linking and shareable URLs. If you want to send someone a link that opens your app in a specific state (aka deep linking), query strings are the simplest way, as shown in the following relative link: `/customers?country=USA`. This means anyone loading that URL will get the customers filtered by USA immediately, without extra clicks. That's especially important for dashboards, filtered views, or search results.

With Blazor apps, state is often lost on refresh or when using the back/forward browser buttons. By putting key parameters in the query string, you automatically preserve state across refreshes (*F5* won't reset your filter) and let the back button take you "back to the previous query." This aligns with normal browser expectations.

Query strings are the lingua franca of web links. If you want another system (say an email campaign, or a deep link from a different site) to pre-load your Blazor component with data, query parameters make that possible without extra server-

side handling.

Route constraints for parameters

Route constraints validate that the data type is correct for a passed parameter. If a potential request with a parameter value violates the constraint, then a match for that route is not made, and other routes will be evaluated instead. If no routes match, then a 404 status code is returned.

If you do not set constraints, then any value is acceptable as a route match, but a data type conversion exception may result when the value is converted into the C# method's expected data type. Some route constraint examples are shown in *Table 4.1*:

Constraint	Example
The <code>IsAnnotated</code> property must be set to a valid Boolean value, for example, <code>TRUE</code> or <code>True</code> .	
The <code>ValidDate</code> property must be a valid date/time value.	
The <code>ValidPercent</code> property must be a valid decimal value.	
The <code>IsValidWeight</code> property must be a valid double value.	
The <code>IsValidBirth</code> property must be a valid float value.	
The <code>Gender</code> property must be a valid GUID value.	
The <code>Category</code> property must be a valid int value.	
The <code>Seconds</code> property must be a valid long value.	

Table 4.1: Route constraint examples

Good practice: Route constraints assume invariant culture, so your URLs must not be localized. For example, always use invariant culture formats to pass date and time parameter values. This means that they expect formats like `yyyy-MM-dd` for dates, `.` as the decimal separator, and so on, regardless of your server or client locale. If you

accidentally pass a localized value (say a German date 31.12.2025 or a French decimal 3,14), Blazor won't match the route because it only understands invariant formats. For example, for 11th November 2025, you must use 2025-11-11. If you use UK style (11-11-2025) or German style (11.11.2025), then the route won't match, and you'll probably get a 404 status code error.

Base component classes

The `OnParametersSet` method is defined by the base class that components inherit from by default, named `ComponentBase`, as shown in the following code:

```
using Microsoft.AspNetCore.Components; public
abstract class ComponentBase : IComponent,
IHandleAfterRender, IHandleEvent { // members
not shown }
```

`ComponentBase` has some useful methods that you can call and override, as shown in *Table 4.2*:

Method (in)		
<code>InvokeAsync()</code>		
Call this method to execute a function on the associated renderer's synchronization context. This avoids the requirement to write thread-synchronizing code when accessing shared resources. Multiple threads are not allowed to access the rendering process at the same time. The use of <code>InvokeAsync</code> means that only one thread will access components at any given moment, which eliminates the need to write thread-locking and synchronization code for shared state.		

Override these methods to invoke code each time the component has been rendered.	RenderAsync	
Override these methods to invoke code after the component has received its initial parameters from its parent in the render tree.	OnInitialParametersAsync	
Override these methods to invoke code after the component has received parameters and the SetAsync have been assigned to properties.	OnParametersAsync	
Override this method to indicate if the component should render.	ShouldRender	
Call this method to cause the component to re-render.	Render	

Table 4.2: Useful methods of ComponentBase

Blazor layouts

Blazor components can have shared layouts in a similar way to MVC views and Razor Pages. You would create a `.razor` component file and make it explicitly inherit from `LayoutComponentBase`, as shown in the following markup:

```
@inherits LayoutComponentBase <div> ... @Body
... </div>
```

The base class has a property named `Body` that you can render in the markup at the correct place within the layout.

You can set a default layout for components in the `Routes.razor` file, as shown highlighted in the following markup:

```
<Router
AppAssembly="@typeof(Program).Assembly">
<Found Context="routeData"> <RouteView
RouteData="@routeData"
DefaultLayout="@typeof(Layout.MainLayout)" />
<FocusOnNavigate RouteData="@routeData"
```

```
Selector="h1" /> </Found> </Router>
```

To explicitly set a layout for a component, use the `@layout` directive, as shown in the following markup:

```
@page "/employees" @layout AlternativeLayout
<div> ... </div>
```

How to navigate to page components

Microsoft provides a dependency service named `NavigationManager` that understands Blazor routing and the `NavLink` component. The `NavigateTo` method is used to go to the specified URL.

In HTML, you use the `<a>` element to define navigation links, as shown in the following markup:

```
<a href="/employees">Employees</a>
```

In Blazor, use the `<NavLink>` component, as shown in the following markup:

```
<NavLink href="/employees">Employees</NavLink>
```

The `NavLink` component is better than an anchor element because it automatically sets its class to `active` if its `href` is a match on the current location URL. If your CSS uses a different class name, then you can set the class name in the `NavLink.ActiveClass` property.

By default, in the matching algorithm, the `href` is a path *prefix*, so if `NavLink` has an `href` of `/employees`, as shown in the preceding code example, then it would match all the following paths and set them all to have the `active` class style:

```
/employees /employees/USA /employees/UK/London
```

To ensure that the matching algorithm only performs matches on *all* of the text in the path, (in other words, there is only a match when the whole complete text matches and not when just part of the path matches), then set the `Match` parameter to `NavLinkMatch.All`, as shown in the following code:

```
<NavLink href="/employees"  
Match="NavLinkMatch.All">Employees</NavLink>
```

If you set other attributes such as `target`, they are passed through to the underlying `<a>` element that is generated.

CSS and JavaScript isolation

Blazor components often need to provide their own CSS to apply styling or JavaScript for activities that cannot be performed purely in C#, like access to browser APIs. To ensure this does not conflict with site-level CSS and JavaScript, Blazor supports CSS and JavaScript isolation.

If you have a component named `Home.razor`, simply create a CSS file named `Home.razor.css`. The styles defined within this file will override any other styles in the project for this component, but not for the rest of the website.

For JavaScript isolation, you do not use a naming convention in the same way as with CSS. Instead, Blazor enables JavaScript isolation using JavaScript modules, imported using the JavaScript interop feature of Blazor, as you will see later in this chapter.

You can read more about JavaScript isolation at the following link: <https://learn.microsoft.com/en-us/aspnet/>


```
core/blazor/javascript-  
interoperability/call-javascript-  
from-dotnet#javascript-isolation-  
in-javascript-modules.
```

Building Blazor components

With ASP.NET Core 8, Blazor introduced a new project template to start a project that supports the most flexible hosting model and all rendering modes. It provides a basic template to run, and a `Weather` component, which shows a table with five rows of random temperatures that uses streaming rendering.

Reviewing the new Blazor project template

First, we will create a Blazor Web App project without interactivity and review its important parts:

1. Use your preferred code editor to create a new project and solution, using the Blazor Web App project template, as defined in the following list:
 - Project template: **Blazor Web App** / `blazor --interactivity None`
 - Project file and folder: `Northwind.Blazor`
 - Solution file and folder: `ModernApps`
 - **Framework: .NET 10.0 (Long Term Support)**
 - **Authentication type: None**
 - **Configure for HTTPS: Selected**
 - **Interactive render mode: None**
 - **Interactivity location: Per page/component**
 - **Include sample pages: Selected**
 - **Do not use top-level statements: Cleared**
 - **Use the .dev.localhost TLD in the application URL:**

Cleared

- **Enlist in Aspire orchestration:** Cleared

If you prefer to use the `dotnet` CLI, then enter the following command at the command prompt or terminal in the `ModernApps` folder: `dotnet new blazor --interactivity None -o Northwind.Blazor.`

Good practice: We have not selected the options to use interactive Blazor WebAssembly or Blazor Server components, so that we can build up your knowledge about how Blazor works step by step. In real-world projects, you are likely to want to select these options from the start. We have also chosen to include sample pages, which you would likely want to clear in a real-world project.

1. Build the `Northwind.Blazor` project.
2. In `Northwind.Blazor.csproj`, note that it is similar to an ASP.NET Core project that uses the Web SDK and targets .NET 10, as shown in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup> <TargetFramework>net10.0</
TargetFramework> <Nullable>enable</
Nullable> <ImplicitUsings>enable</
ImplicitUsings>
  <BlazorDisableThrowNavigationException>
true</
```

```
BlazorDisableThrowNavigationException> </  
PropertyGroup> </Project>
```

Navigation in the app is handled by the Blazor router (`<Router>`). If Blazor can't handle the navigation (say the URL points outside the app's routing scope or something went wrong internally), Blazor throws a special exception called `NavigationException`. That exception is not really an error in the traditional sense. It's a control signal to say, "This navigation should be handed off to the browser instead of being handled internally." Often, these unhelpful exceptions are logged in output windows when doing normal navigation (like clicking an external link or refreshing a page). This cluttered logs and made it seem like the app was failing when it wasn't. If `<BlazorDisableThrowNavigationException>` is set to `true`, Blazor avoids throwing and instead uses a quieter mechanism.

1. In the `Northwind.Blazor` project, in `Program.cs`, note that the statements enable the ASP.NET Core service collection and HTTP pipeline, with Blazor-specific statements to add Razor components and then map them based on your Blazor routing configuration, and enable static asset compression introduced with .NET 9 using the `MapStaticAssets` method, as shown highlighted in the following code:

```
using Northwind.Blazor.Components; // To
```

```
use your project's components. var builder
= WebApplication.CreateBuilder(args); //
Add services to the container.
builder.Services.AddRazorComponents(); var
app = builder.Build(); // Configure the
HTTP request pipeline. if (!
app.Environment.IsDevelopment()) {
app.UseExceptionHandler("/Error",
createScopeForErrors: true); // The
default HSTS value is 30 days. You may
want to change this for production
scenarios, see https://aka.ms/aspnetcore-
hsts. app.UseHsts(); }
app.UseStatusCodePagesWithReExecute("/not-
found", createScopeForStatusCodePages:
true); app.UseHttpsRedirection();
app.UseAntiforgery();
app.MapStaticAssets();
app.MapRazorComponents<App>(); app.Run();
```

Normally, when ASP.NET Core returns an HTTP status code like 404 Not Found or 500 Server Error, it just writes a plain empty response (or a very simple text page). That's not a good user experience in a Blazor app. The `UseStatusCodePagesWithReExecute` middleware changes that behavior. It tells ASP.NET Core, "Whenever you're about to return a status code like 404, instead of sending it straight to the browser, re-execute the pipeline as if the user had navigated to `/not-found`." So, the

request is internally rerouted to /not-found where you can render a proper Blazor page component. In a Blazor app, this means you can use Razor layouts, components, branding, and so on, for your error page.

1. In the Northwind.Blazor project, expand the Properties folder, open the launchSettings.json file, and for the applicationUrl setting of the https profile, change the port numbers to 5041 for https and 5042 for http, as shown in the following setting:

```
"applicationUrl": "https://  
localhost:5041;http://localhost:5042",
```

2. Save the changes to the launchSettings.json file.
3. In the Northwind.Blazor project, in the Components folder, open App.razor, as shown in the following markup:

```
<!DOCTYPE html> <html lang="en"> <head>  
<meta charset="utf-8" /> <meta  
name="viewport" content="width=device-  
width, initial-scale=1.0" /> <base  
href="/" /> <ResourcePreloader /> <link  
rel="stylesheet" href= "@Assets["lib/  
bootstrap/dist/css/bootstrap.min.css"]" />  
<link rel="stylesheet"  
href="@Assets["app.css"]" /> <link  
rel="stylesheet" href=  
"@Assets["Northwind.Blazor.styles.css"]" /  
> <ImportMap /> <link rel="icon"  
type="image/png" href="favicon.png" />
```

```
<HeadOutlet /> </head> <body> <Routes />
<script src="@Assets["_framework/
blazor.web.js"]"></script> </body> </html>
```

Note the following:

- Assets are referenced using the `@Assets` property of `ComponentBase`, which resolves the fingerprinted URL for a given asset. This must be used when you use the `MapStaticAssets` middleware in `Program.cs` to render the correct path to the compressed asset.
- A `<ResourcePreloader />` Blazor component that renders preload link elements for resources. This permits the app base path configuration (`<base href="..." />`) to correctly identify the app's root.
- An `<ImportMap />` Blazor component to represent an import map element (`<script type="importmap"></script>`) that defines the import map for module scripts. You can learn about import maps at the following link: <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script/type/importmap>.
- A `<HeadOutlet />` Blazor component to inject additional content into the `<head>` section. This is one of the built-in components available in all Blazor projects.
- A `<Routes />` Blazor component to define the custom routes in this project. This component can be completely customized by the developer because it is part of the current project, in a file named `Routes.razor`.
- A script block for `blazor.web.js` that manages communication back to the server for Blazor's dynamic features, like downloading WebAssembly components in the background and later switching from server-side to client-side component

execution.

1. In the `Components` folder, in `Routes.razor`, note that a `<Router>` enables routing for all Blazor components found in the current assembly, and that if a matching route is found, then `RouteView` is executed, which sets the default layout for the component to `MainLayout` and passes any route data parameters to the component. For that component, the first `<h1>` element in it will get the focus, as shown in the following code:

```
<Router
AppAssembly="@typeof(Program).Assembly"
NotFoundPage="typeof(Pages.NotFound)">
  <Found Context="routeData"> <RouteView
RouteData="@routeData"
DefaultLayout="@typeof(Layout.MainLayout)"
/> <FocusOnNavigate RouteData="@routeData"
Selector="h1" /> </Found> </Router>
```

2. In the `Components` folder, in `_Imports.razor`, note that this file imports some useful namespaces for use in all your custom Blazor components, as shown in the following code:

```
@using System.Net.Http @using
System.Net.Http.Json @using
Microsoft.AspNetCore.Components.Forms
@using
Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using static
Microsoft.AspNetCore.Components.Web.RenderMode
@using
Microsoft.AspNetCore.Components.Web.Virtualization
```

```
@using Microsoft.JSInterop @using
Northwind.Blazor // To use your project
code. @using Northwind.Blazor.Components
// To use your components. @using
Northwind.Blazor.Components.Layout // To
use your layouts.
```

3. In the `Components\Layout` folder, note that `MainLayout.razor` defines a `<div>` for a sidebar, containing a navigation menu that is implemented by the `NavMenu.razor` component file in this project, and HTML5 elements like `<main>` and `<article>` for the content, as shown in the following markup:

```
@inherits LayoutComponentBase <div
class="page"> <div class="sidebar">
<NavMenu /> </div> <main> <div class="top-
row px-4"> <a href="https://
learn.microsoft.com/aspnet/core/"
target="_blank">About</a> </div> <article
class="content px-4"> @Body </article> </
main> </div>
```

4. In the `Components\Layout` folder, open `NavMenu.razor`, as shown in the following markup:

```
<div class="top-row ps-3 navbar navbar-
dark"> <div class="container-fluid"> <a
class="navbar-brand"
href="">Northwind.Blazor</a> </div> </div>
<input type="checkbox" title="Navigation
menu" class="navbar-toggler" /> <div
class="nav-scrollable"
onclick="document.querySelector('.navbar-
```



```
toggler').click() "> <nav class="flex-
column"> <div class="nav-item px-3">
<NavLink class="nav-link" href=""
Match="NavLinkMatch.All"> <span class="bi
bi-house-door-fill-nav-menu" aria-
hidden="true"></span> Home </NavLink> </
div> <div class="nav-item px-3"> <NavLink
class="nav-link" href="weather"> <span
class="bi bi-list-nested-nav-menu" aria-
hidden="true"> </span> Weather </NavLink>
</div> </nav> </div>
```

Note the following:

- The `NavMenu` component does not have a `@page` directive because it does not use a shared layout or render as a page.
- It uses Bootstrap to provide a menu of choices that responsively adapts to the width of the viewport. It will collapse into a hamburger menu when there is not enough horizontal space, and then the visitor can toggle the navigation on and off.
- There are currently two menu items: **Home** and **Weather**. We will add more throughout this chapter.

1. In the `Components\Pages` folder, in `Home.razor`, note the `@page` directive that configures a route for the root path to go to this page component, and then change the heading from `world` to `Blazor Full Stack`, as shown highlighted in the following markup:

```
@page "/" <PageTitle>Home</PageTitle>
<h1>Hello, Blazor Full Stack!</h1> Welcome
to your new app.
```

2. In the `Components\Pages` folder, in `NotFound.razor`, note that

when a user requests something invalid like `/customers/9999` (and your app has no such route), the ASP.NET Core middleware will by default re-execute the pipeline to `/not-found`, the request gets mapped to this page (`@page "/not-found"`), it's rendered inside `MainLayout`, and the user sees a styled “Not Found” page, not a blank error, as shown in the following markup:

```
@page "/not-found" @layout MainLayout
<h3>Not Found</h3> <p>Sorry, the content
you are looking for does not exist.</p>
```

3. Start the `Northwind.Blazor` project, using its `https` profile without debugging:

- If you are using Visual Studio, then in **Solution Explorer**, select the `Northwind.Blazor` project to make it active. In the Visual Studio toolbar, select the `https` profile as the **Startup Project**, and **Google Chrome** as the **Web Browser**.
- If you are using VS Code, then at the command line or terminal, enter the following command:

```
dotnet run --launch-profile https
```

6. In Chrome, note the left-side navigation and the home page component, as shown in *Figure 4.1*:

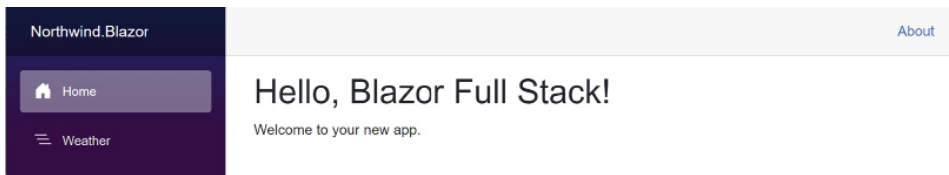


Figure 4.1: A simple web page implemented as a Blazor page component

1. Finally, close the browser, and at the command prompt or terminal, press *Ctrl + C* to shut down the web server.

Using Bootstrap icons

The older Blazor project templates included all the Bootstrap icons. In the new project template, only three icons are defined using **Scalable Vector Graphics (SVG)**. Let's see how the team defined those icons, and then add some more for our own use:

1. In the `Components\Layout` folder, in the CSS stylesheet file named `NavMenu.razor.css`, find the text `bi-house`, and note the three icons defined using SVG, as partially shown in the following code:

```
.bi-house-door-fill-nav-menu { background-image: url("data:image/svg+xml,..."); }
.bi-plus-square-fill-nav-menu { background-image: url("data:image/svg+xml,..."); }
.bi-list-nested-nav-menu { background-image: url("data:image/svg+xml,..."); }
```

2. In your favorite browser, navigate to: <https://icons.ionicons.com/> (you must include the ending / to go to the correct page!), and note that **Bootstrap Icons** have an MIT license and contain more than 2,000 icons.
3. In the **Filter Icons** box (not the **Search icons** box!), enter `globe`, and note that multiple globe icons are found, as shown in *Figure 4.2*:

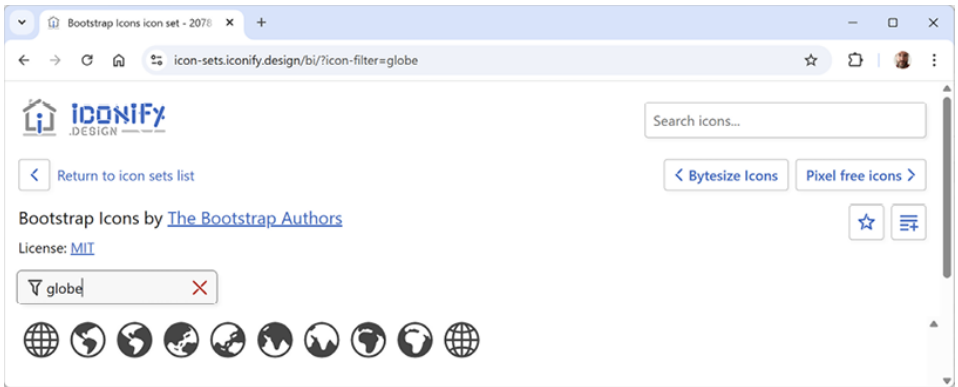


Figure 4.2: Searching for globe icons

1. Click the first globe, scroll down the page, and click the **SVG as data: URI** button. Note that you could copy and paste the definition of this icon for use in the CSS stylesheet, but you do not need to because I have already created a CSS file for you to use, with multiple icons defined for you to use in your Blazor project.
2. In your favorite browser, navigate to: <https://github.com/markjprice/apps-services-net10/blob/main/code/ModernApps/Northwind.Blazor/wwwroot/icons.css>, download the file, and save it in your own project in its `wwwroot` folder.
3. In the `Components` folder, in the `App.razor` component, in the `<head>`, add a `<link>` element to reference the `icons.css` stylesheet, as shown in the following markup:

```
<link rel="stylesheet"
href="@Assets["icons.css"]" />
```

4. Save and close the file.

Referencing an EF Core class library and

registering a data context

We will reference the EF Core model that you created in [Chapter 1](#), *Introducing Apps and Services with .NET*:

1. In the `Northwind.Blazor.csproj` project file, treat warnings as errors, and add a project reference to the Northwind database context project, as shown in the following markup:

```
<ItemGroup> <ProjectReference Include="..\
  \Northwind.DataContext
  \Northwind.DataContext.csproj" /> </
ItemGroup>
```

The `Include` path must not have a line break.

1. Build the `Northwind.Blazor` project.
2. In the `Components` folder, in `_Imports.razor`, import the namespaces to use asynchronous methods with EF Core and to work with the Northwind entity models, as shown in the following code:

```
@using Microsoft.EntityFrameworkCore @* To
useToListAsync method. *@ @using
Northwind.EntityModels @* To use
NorthwindContext and so on. *@
```

Importing the namespaces here means we do not have to import them at the top of `.razor` files. The `_Imports.razor` file only applies to `.razor` files. If you use code-behind `.cs` files to implement

component code, then they must have namespaces imported separately, or use global usings to implicitly import the namespace. Note the statement to statically import the render mode type:

```
@using static
```

```
Microsoft.AspNetCore.Components.Web
```

1. In `Program.cs`, import the namespace to use the `AddNorthwindContext` extension method, as shown in the following code:

```
using Northwind.EntityModels; // To use
AddNorthwindContext method.
```

2. In the section that adds services to the container, add a statement that registers `NorthwindContext` as a service, as shown in the following code:

```
builder.Services.AddNorthwindContext();
```

Building a static server rendered component for data

Next, we will add a component that can do the same job as a Razor Page or Razor View in a traditional ASP.NET Core website. It will not have any interactivity that requires the component to execute on the server or the client.

It will allow the visitor to see a table of products from the Northwind database:

1. In the `Components\Pages` folder, add a new file named `Products.razor`. In Visual Studio, the project item template is

named **Razor Component**. In Rider, the project item template is named **Blazor Component**.

Good practice: Component filenames must start with an uppercase letter, or you will have compile errors!

1. In `Products.razor`, set the route to `/products`, inject a Northwind data context, define a table to render products, and write a code block to get the products when the page has initialized, as shown in the following markup:

```
@page "/products" @inject NorthwindContext
db <h1>Products</h1> <table class="table">
<thead> <tr> <th>Product ID</th>
<th>Product Name</th> <th>Unit Price</th>
</tr> </thead> <tbody> @if ((products is
null) || (products.Count == 0)) { <tr><td
colspan="4">No products found.</td></tr> }
else { @foreach (Product p in products) {
<tr> <td>@p.ProductId</td>
<td>@p.ProductName</td>
<td>@p.UnitPrice.HasValue ?
p.UnitPrice.Value.ToString("C") : "n/a"</
td> </tr> } } </tbody> </table> @code {
private List<Product>? products; protected
override async Task OnInitializedAsync() {
products = await
db.Products.ToListAsync(); } }
```

2. In the `Components\Layout` folder, in `NavMenu.razor`, after the menu item to navigate to the home page, add a menu item to navigate to the products page, as shown in the following markup:

```
<div class="nav-item px-3"> <NavLink  
class="nav-link" href="products"> <span  
class="bi bi-globe" aria-hidden="true"></  
span> Products </NavLink> </div>
```

3. If your database server is not running (for example, because you are hosting it in Docker, a virtual machine, or the cloud), then make sure to start it.
4. Start the `Northwind.Blazor` project, using its `https` profile without debugging.
5. In the left-side navigation, click **Products**, and note the table of products.
6. Close the browser and shut down the web server.

Building a component with server interactivity

Next, we will add a component that requires some interactivity, so we will enable Blazor with SignalR to dynamically update the browser DOM live at runtime:

1. In the `Components` folder, add a new file named `Counter.razor`.
2. In `Counter.razor`, define a label to render the current value of the counter number, a button to increment it, and a code block to store the current counter value with a click event handler, to increment the number, as shown in the following markup:

```
<h3>Counter: @CounterValue</h3> <button  
id="buttonIncrement"  
@onclick="IncrementCounter" class="btn  
btn-outline-primary">Increment</button>  
@code { public int CounterValue { get;  
set; } = 0; public void IncrementCounter()
```



```
{ CounterValue++; } }
```

Note that this component will not act as a page, so it is stored in the `Component` folder instead of the `Components` \Pages folder, we do not decorate it with the `@page` directive, or define a route to the component. It will only be used embedded in some other component.

1. In the `Components\Pages` folder, in `Home.razor`, at the bottom of the page, render a horizontal rule and the counter component, as shown in the following markup:

```
<hr /> <Counter />
```

2. Start the `Northwind.Blazor` project, using its `https` profile without debugging.
3. On the home page, click the button, and note that nothing happens. When you create a project using the Blazor Web App template with the `--interactivity None` switch or the **Interactive render mode** set to **None**, no component interactivity is enabled in a Blazor Web App project.
4. Close the browser and shut down the web server.
5. In `Program.cs`, at the end of the statement that adds Razor components, add a call to a method to add interactive server components, as shown highlighted in the following code:

```
builder.Services.AddRazorComponents()  
    .AddInteractiveServerComponents();
```

6. In `Program.cs`, at the end of the statement that maps Razor components, add a call to a method to add interactive server render mode, as shown highlighted in the following code:

```
app.MapRazorComponents<App> ()  
    .AddInteractiveServerRenderMode ();
```

7. In the `Components` folder, in `Counter.razor`, at the top of the file, add a directive to set render mode to interactive server, as shown in the following markup:

```
@rendermode InteractiveServer
```

8. Start the `Northwind.Blazor` project, using its `https` profile without debugging.
9. In **Developer Tools**, click the **Console** tab, and note the `blazor.web.js` files establishes a WebSocket connection, as shown in the following output:

```
[2025-10-20T11:25:52.498Z] Information:  
Normalizing '_blazor' to 'https://  
localhost:5041/_blazor'.  
[2025-10-20T11:25:52.675Z] Information:  
WebSocket connected to wss://  
localhost:5041/_blazor?  
id=j6Fc0Mbay_jWkZTWfIqs_w.
```

10. In **Developer Tools**, click the **Network** tab, click **WS** to filter by WebSockets, and refresh the home page.
11. On the home page, click the **Increment** button, note the counter increments, note the `_blazor?id=...` request, select the request for

`_blazor?id=...`, and click the **Initiator** tab, and note that the initiator was the `blazor.web.js` file that is added to all pages by the `App.razor` file, as shown in *Figure 4.3*:

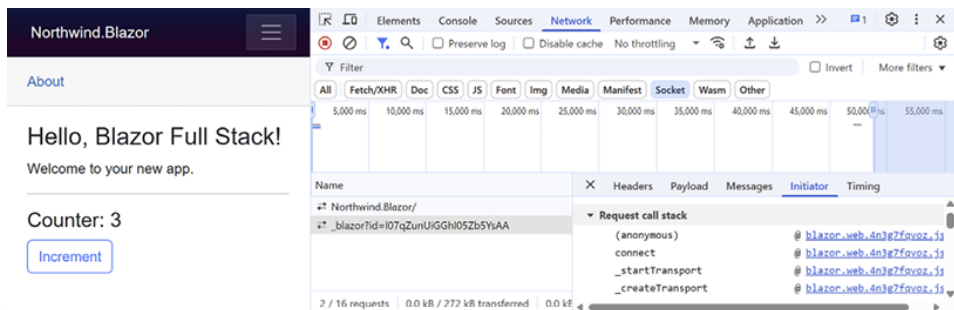


Figure 4.3: Blazor.web.js requests to SignalR on the server to update the DOM for live interactivity

1. Close the browser and shut down the web server.

Building a Blazor progress bar component

In this section, we will build a component to provide a progress bar. It will use Bootstrap classes to set a striped light blue color, with options to animate the bar and show the current value of the progress as a percentage:

1. In the `Northwind.Blazor` project, in the `Components` folder, add a new file named `ProgressBar.razor`.
2. In `ProgressBar.razor`, add statements to render `<div>` elements that use Bootstrap classes to define a progress bar with bindable parameters, setting various properties, as shown in the following markup:

```
@rendermode InteractiveServer <div
class="progress"> <div class="progress-bar
progress-bar-striped bg-info @(IsAnimated
? " progress-bar-animated" : "")"
role="progressbar" aria-label="@LabelText"
```

```

style="width: @Value%" aria-
valuenow="@Value" aria-valuemin="@Minimum"
aria-valuemax="@Maximum"> @(ShowValue ?
Value + "%" : "") </div> </div> @code {
[Parameter] public int Value { get; set; }
= 0; [Parameter] public int Minimum { get;
set; } = 0; [Parameter] public int Maximum
{ get; set; } = 100; [Parameter] public
bool IsAnimated { get; set; } = false;
[Parameter] public bool ShowValue { get;
set; } = false; [Parameter] public string?
LabelText { get; set; } = "Progress bar";
}

```

3. In the Components\Pages folder, in Home.razor, at the bottom of the file, add statements to add a horizontal rule and define a Bootstrap row with two equal columns, and add a <ProgressBar> component set to 25%, as shown in the following markup:

```

<hr /> <div class="row"> <div class="col">
<div class="alert alert-info">
<h4>Progress of database deletion</h4>
<ProgressBar Value="25" IsAnimated="true"
ShowValue="true" LabelText="Progress of
database deletion" /> </div> </div> <div
class="col"> More components coming soon.
</div> </div>

```

4. Start the Northind.Blazor project, using its https profile without debugging.
5. Note the progress bar that shows the progress of the (simulated!) database deletion.
6. Close the browser and shut down the web server.

Building a Blazor dialog box component

In this section, we will build a component to provide a popup dialog box for interaction with the website visitor. It will use Bootstrap classes to define a button that, when clicked, shows a dialog box with two buttons with configurable labels.

By default, the Blazor Web App project template uses a local copy of Bootstrap 5.3.3 but only the CSS part. We will need to add a script tag to add the JavaScript parts of Bootstrap. We may as well upgrade to the latest version of Bootstrap and use the CDN version too.

You can find the latest CDN links at the following link: <https://getbootstrap.com/docs/5.3/getting-started/introduction/#cdn-links>.

The component will also define two event callbacks that can be handled by the parent to customize which code executes when the two buttons are clicked:

1. In `App.razor`, comment out the `<link>` to the local CSS file and add a reference to the latest CDN version, as shown highlighted in the following markup:

```
@**@<link rel="stylesheet"
href="@Assets["lib/bootstrap/dist/css/
bootstrap.min.css"]" /> <link
rel="stylesheet" href="https://
cdn.jsdelivr.net/npm/bootstrap@5.3.8/dist/
css/bootstrap.min.css" integrity="sha384-
sRI14kxILFvY47J16cr9ZwB07vP4J8+LH7qKQnuqkuIAvNWLz
crossorigin="anonymous">
```

2. In `App.razor`, after the `<script>` tag for Blazor, add a

<script> to the latest CDN version, and suppress error RZ/BL9992, as shown in the following markup:

```
<script src="https://cdn.jsdelivr.net/npm/
bootstrap@5.3.8/dist/js/
bootstrap.bundle.min.js"
integrity="sha384-
FKyoeFForCGlyvwx9Hj09JcYn3nv7wiPVlz7YYwJrWVcXK/
BmnVDxM+D2scQbITxI"
crossorigin="anonymous" suppress-
error="BL9992"></script>
```

We suppress the error BL9992 (also referred to as RZ9992), which warns that “Script tags should not be placed inside components because they cannot be updated dynamically.” For more information, see <https://aka.ms/AAe3qu3>.

1. In the Northwind.Blazor project, in the Components folder, add a new file named DialogBox.razor.
2. In DialogBox.razor, add statements to render <div> elements that use Bootstrap classes to define a button and modal dialog box with bindable parameters, setting various properties, as shown in the following markup:

```
@rendermode InteractiveServer <!-- Button
to show the dialog box. --> <button
type="button" class="btn btn-primary"
data-bs-toggle="modal" data-bs-
target="#dialogBox"> @DialogTitle </
button> <!-- Dialog box to popup. --> <div
```

```

class="modal fade" id="dialogBox" data-bs-
backdrop="static" data-bs-keyboard="false"
tabindex="-1" aria-
labelledby="dialogBoxLabel" aria-
hidden="true"> <div class="modal-dialog">
<div class="modal-content"> <div
class="modal-header"> <h5 class="modal-
title" id="dialogBoxLabel">@DialogTitle</
h5> <button type="button" class="btn-
close" data-bs-dismiss="modal" aria-
label="Close"></button> </div> <div
class="modal-body"> @ChildContent </div>
<div class="modal-footer"> <button
type="button" class="btn btn-primary"
@onclick="OnClickPrimary">
@PrimaryButtonText </button> <button
type="button" class="btn btn-secondary"
data-bs-dismiss="modal"
@onclick="OnClickSecondary">
@SecondaryButtonText </button> </div> </
div> </div> </div> @code { [Parameter]
public string? DialogTitle { get; set; }
// ChildContent is a special name that is
set automatically by any // markup content
within the component begin and end
elements. [Parameter] public
RenderFragment? ChildContent { get; set; }
[Parameter] public string?
PrimaryButtonText { get; set; } = "OK";
[Parameter] public
EventCallback<MouseEventArgs>
OnClickPrimary { get; set; } [Parameter]
public string? SecondaryButtonText { get;
set; } = "Cancel"; [Parameter] public
EventCallback<MouseEventArgs>

```

```
OnClickSecondary { get; set; } }
```

Note that the two buttons have default text values of `OK` and `Cancel`, and they both have event callback parameters that will have information about the mouse pointer passed as event arguments. Also, note the button with `class="btn-close"` that visually appears as the **X** button in the top-right corner to close the dialog.

1. In the `Components\Pages` folder, in `Home.razor`, at the top of the file, add statements to set render mode as the interactive server, as shown in the following markup:

```
@rendermode InteractiveServer
```

2. In the `Components\Pages` folder, in `Home.razor`, near the bottom of the file, replace the text `More components coming soon` with statements to add a `<DialogBox>` component that sets the two button labels to `Yes` and `No`. Then, at the bottom of the file, add a Razor code block to define event handlers for the two click events that output which button was clicked and the current position of the mouse pointer, as shown highlighted in the following markup:

```
<div class="col"> <DialogBox  
DialogTitle="Delete Database"  
PrimaryButtonText="Yes"  
OnClickPrimary="Yes_Click"  
SecondaryButtonText="No"
```



```

OnClickSecondary="No_Click"> Are you sure
you want to delete the entire database?
Really? </DialogBox> </div> </div> @code {
private void Yes_Click(MouseEventArgs e) {
Console.WriteLine("User clicked 'Primary'
button at ({0}, {1}).", arg0: e.ClientX,
arg1: e.ClientY); } private void
No_Click(MouseEventArgs e) {
Console.WriteLine("User clicked
'Secondary' button at ({0}, {1}).", arg0:
e.ClientX, arg1: e.ClientY); } }

```

Any content between the `<DialogBox>` and `</DialogBox>` elements is automatically set as the `ChildContent` property.

1. Start the `Northwind.Blazor` project, using its `https` profile without debugging.
2. Click the **Delete Database** button, and note the modal dialog box that pops up, as shown in *Figure 4.4*:

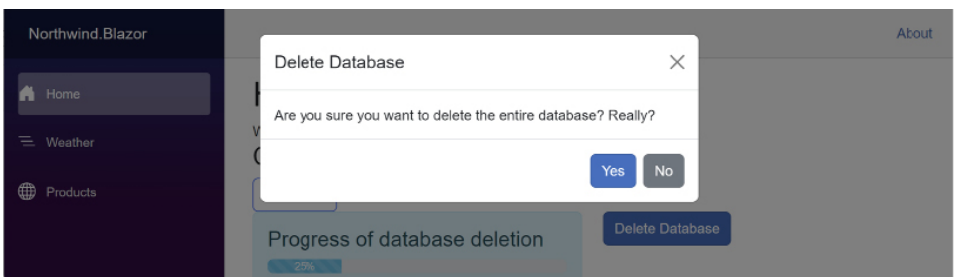


Figure 4.4: A pop-up modal dialog box that a Blazor component built using Bootstrap

1. Arrange the command prompt or terminal and the browser window so that you can see both.
2. In the **Delete Database** dialog box, click the **Yes** button and **No** button a

few times (clicking the **No** button or the **x** button will close the dialog, so click the **Delete Database** button again to reshown the dialog box), and note the messages written to the console, as shown in the following output:

```
User clicked 'Primary' button at (686, 179). User clicked 'Secondary' button at (734, 187). User clicked 'Primary' button at (673, 192). User clicked 'Primary' button at (701, 174). User clicked 'Secondary' button at (748, 184).
```

JavaScript in the client displays the Bootstrap dialog box. Clicks on the buttons trigger the SignalR connection over WebSockets to execute the component event handling code, executing on the server.

1. Close the browser and shut down the web server.

You can read more about the supported event arguments at the following link:
<https://learn.microsoft.com/en-us/aspnet/core/blazor/components/event-handling#event-arguments>.

Building a Blazor alert component

In this section, we will build a component to provide alerts to show messages to

the website visitor. It will use Bootstrap classes to define a colorful area for the message, which can be dismissed. The message, title, icon, and color theme can be configured:

1. In the `Northwind.Blazor` project, in the `Components` folder, add a new class file named `Bootstrap.Constants.cs`.
2. In `Bootstrap.Constants.cs`, add statements to define some static classes with `string` constant values for common Bootstrap color themes and icons, as shown in the following code:

```
namespace Northwind.Blazor.Components;
public static class BootstrapColors {
    public const string Primary = "primary";
    public const string Secondary =
        "secondary"; public const string Danger =
        "danger"; public const string Warning =
        "warning"; public const string Success =
        "success"; public const string Info =
        "info"; } public static class
BootstrapIcons { public const string Globe
    = "bi bi-globe"; public const string
    GlobeEmea = "bi bi-globe-europe-africa";
    public const string Pencil = "bi bi-
    pencil"; public const string Trash = "bi
    bi-trash"; public const string PlusSquare
    = "bi bi-plus-square"; public const string
    InfoCircle = "bi bi-info-circle"; public
    const string ExclamationTriangleFill = "bi
    bi-exclamation-triangle-fill"; }
```

3. In the `Components` folder, add a new file named `Alert.razor`.
4. In `Alert.razor`, add statements to render `<div>` elements that use Bootstrap classes to define a `<div>` with bindable parameters, setting various properties, as shown in the following markup:

```
@rendermode InteractiveServer <div
class="alert alert-@ColorTheme d-flex
align-items-center @(IsDismissable ? "
alert-dismissible fade show" : "")"
role="alert"> <div> <h4 class="alert-
heading"><span class="@Icon" aria-
hidden="true"> </span> @Title</h4>
@Message @if (IsDismissable) { <button
type="button" class="btn-close" data-bs-
dismiss="alert" aria-label="Close"></
button> } </div> </div> @code {
[Parameter] public bool IsDismissable {
get; set; } = true; [Parameter] public
string ColorTheme { get; set; } =
BootstrapColors.Primary; [Parameter]
public string Icon { get; set; } =
BootstrapIcons.InfoCircle; [Parameter]
public string? Title { get; set; }
[Parameter] public string? Message { get;
set; } }
```

5. In the Components\Pages folder, in Home.razor, add an Alert element below the <DialogBox> element, as shown in the following markup:

```
<Alert IsDismissable="true"
Icon="@ (BootstrapIcons.ExclamationTriangleFill)"
ColorTheme="@ (BootstrapColors.Warning)"
Title="Warning" Message="Deleting the
database cannot be undone." />
```

6. Start the Northwind.Blazor project, using its https profile without debugging.

7. On the home page, note the warning alert, as shown in *Figure 4.5*:

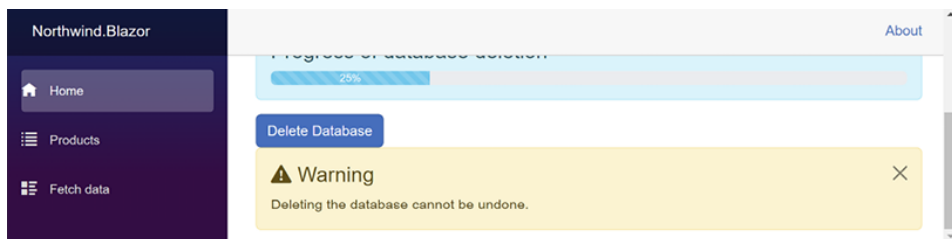


Figure 4.5: The alert component with a dismiss button

1. Click the close button to dismiss the warning.
2. Close the browser and shut down the web server.

Building a Blazor data component

In this section, we will build a component that will list, create, and edit employees in the Northwind database.

We will add the new component to the existing Blazor project:

1. In the `Northwind.Blazor` project, in the `Components\Pages` folder, add a new file named `Employees.razor`.
2. Add statements to output a heading for the `Employees` component and define a code block, defining a property to store the name of a country, as shown highlighted in the following markup:

```
@rendermode InteractiveServer
<h1>Employees
@(string.IsNullOrWhiteSpace(Country) ?
"Worldwide" : "in " + Country)</h1> @code
{ [Parameter] public string? Country {
  get; set; } }
```

3. In the `Components\Pages` folder, in `Home.razor`, after the welcome message, instantiate the `Employees` component twice: once

setting `USA` as the `Country` parameter, and once without setting the country, as shown highlighted in the following markup:

```
<h1>Hello, Blazor Full Stack!</h1> Welcome  
to your new app. <Employees Country="USA"  
/> <Employees />
```

4. Start the `Northwind.Blazor` project, using its `https` profile without debugging.
5. Start Chrome, navigate to `https://localhost:5041/`, and note the `Employees` components, as shown in *Figure 4.6*:

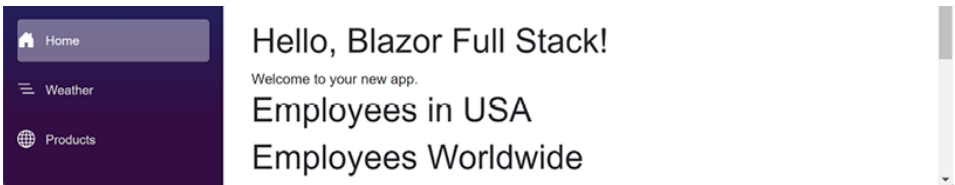


Figure 4.6: The `Employees` component, with the `Country` parameter set to `USA` and without parameters set

1. Close the browser and shut down the web server.

Making the component a routable page component

It is simple to turn this component into a routable page component with a route parameter for the country:

1. In the `Components\Pages` folder, in the `Home.razor` component, remove the two `<Employee>` elements because we will now use them as pages.
2. In the `Components\Pages` folder, in the `Employees.razor` component, add a statement at the top of the file to register `/employees` as its route, with an optional country route parameter, as

shown highlighted in the following markup:

```
@rendermode InteractiveServer @page "/"
employees/{country?}"
```

3. In the `Components\Layout` folder, in `NavMenu.razor`, add list item elements after **Products** to navigate to show employees worldwide and in the USA or UK, as shown in the following markup:

```
<div class="nav-item px-3"> <NavLink
class="nav-link" href="employees"
Match="NavLinkMatch.All"> <span class="bi
bi-globe" aria-hidden="true"></span>
Worldwide </NavLink> </div> <div
class="nav-item px-3"> <NavLink
class="nav-link" href="employees/USA">
<span class="bi bi-people" aria-
hidden="true"></span> Employees in USA </
NavLink> </div> <div class="nav-item
px-3"> <NavLink class="nav-link"
href="employees/UK"> <span class="bi bi-
person" aria-hidden="true"></span>
Employees in UK </NavLink> </div>
```

4. Start the `Northwind.Blazor` project, using its `https` profile without debugging.
5. Start Chrome and navigate to `https://localhost:5041/`.
6. In the left navigation menu, click **Employees in USA**. Note that the country name is correctly passed to the page component and that the component uses the same shared layout as the other page components, like `Home.razor`. Also note the URL: `https://localhost:5041/employees/USA`.

7. Close Chrome and shut down the web server.

Getting entities into a Blazor component

Now, we can add the functionality to the entity component to call the web service:

1. In the `Northwind.Blazor` project, in its project file, add a package reference for QuickGrid, as shown in the following markup:

```
<ItemGroup> <PackageReference  
  Include="Microsoft.AspNetCore.Components.QuickGrid  
</ItemGroup>
```

2. Build the project to restore packages.
3. In the `Components` folder, in `_Imports.razor`, import the namespace needed to work with QuickGrid, as shown in the following markup:

```
@using  
Microsoft.AspNetCore.Components.QuickGrid
```

4. In the `Components\Pages` folder, in `Employees.razor`, add statements to output a grid of either all employees or employees in the specific country, as shown highlighted in the following code:

```
@rendermode InteractiveServer @page "/"  
employees/{country?}" @inject  
NorthwindContext db <h1> Employees  
<string.IsNullOrEmpty(Country) ?  
"Worldwide" : "in " + Country) </h1>  
<QuickGrid Items="@employees" Class="table
```



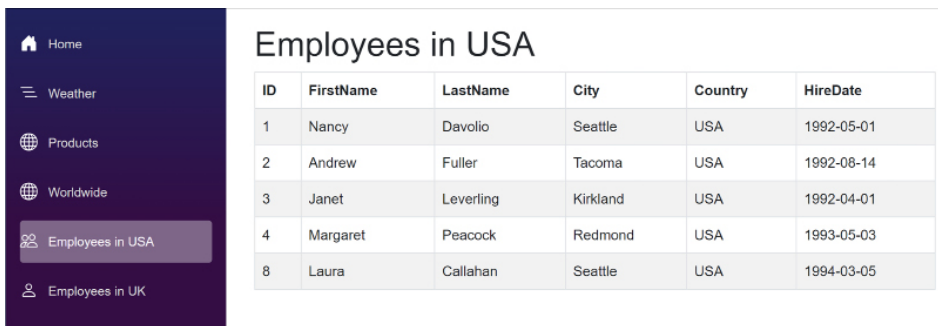
```

table-striped table-bordered">
<PropertyColumn Property="@ (emp =>
emp.EmployeeId) " Title="ID" />
<PropertyColumn Property="@ (emp =>
emp.FirstName) " /> <PropertyColumn
Property="@ (emp => emp.LastName) " />
<PropertyColumn Property="@ (emp =>
emp.City) " /> <PropertyColumn
Property="@ (emp => emp.Country) " />
<PropertyColumn Property="@ (emp =>
emp.HireDate) " Format="yyyy-MM-dd" /> </
QuickGrid> @code { [Parameter] public
string? Country { get; set; } // QuickGrid
works best if it binds to an IQueryable<T>
sequence. private IQueryable<Employee>?
employees; protected override async Task
OnParametersSetAsync() { Employee[]?
employeesArray = null; if (Country is
null) { employees = (await db.Employees
.ToArrayAsync()).AsQueryable(); } else {
employees = (await db.Employees .Where(emp
=> emp.Country == Country)
.ToArrayAsync()).AsQueryable(); } } }

```

5. If your database server is not running (for example, because you are hosting it in Docker, a virtual machine, or in the cloud), then make sure to start it.
6. Start the `Northwind.MinimalApi.Service` project, using its `https` profile without debugging.
7. Start the `Northwind.Blazor` project, using its `https` profile without debugging.
8. Start Chrome, and navigate to `https://localhost:5041/`.
9. In the left navigation menu, click **Employees in USA**, and note that the grid of employees loads from the SQL Server database and renders in the

web page, as shown in *Figure 4.7*:



The screenshot shows a web application with a dark purple navigation menu on the left and a white content area on the right. The navigation menu has links for Home, Weather, Products, Worldwide, Employees in USA (selected), and Employees in UK. The content area displays a table titled 'Employees in USA' with columns for ID, FirstName, LastName, City, Country, and HireDate. The table contains five rows of employee data.

ID	FirstName	LastName	City	Country	HireDate
1	Nancy	Davolio	Seattle	USA	1992-05-01
2	Andrew	Fuller	Tacoma	USA	1992-08-14
3	Janet	Leverling	Kirkland	USA	1992-04-01
4	Margaret	Peacock	Redmond	USA	1993-05-03
8	Laura	Callahan	Seattle	USA	1994-03-05

Figure 4.7: The grid of employees in the USA

1. In the left navigation menu, click **Employees in UK**, and note that the grid of employees is filtered to only show employees in the UK.
2. Close Chrome and shut down the web server.

Practicing and exploring

Test your knowledge and understanding by answering some questions, getting some hands-on practice, and exploring this chapter’s topics with deeper research.

Exercise 4.1 – Online material

Online material can be extra content written by me for this book, or it can be references to content created by Microsoft or third parties.

Explore Blazor WebAssembly topics

To read about topics related specifically to Blazor WebAssembly projects, I have written an online-only section, found at the following link: <https://github.com/markjprice/apps-services-net10/blob/main/docs/ch04-blazor-webassembly.md>.

Explore Progressive Web Apps with Blazor

To read about topics related specifically to Blazor PWA support, I have written an online-only section, found at the following link: <https://github.com/markjprice/apps-services-net10/blob/main/docs/ch04-blazor-pwa.md>.

Leveraging Open Source Blazor Component Libraries

To learn how to use some common Blazor open-source components, I have written an online-only section, found at the following link: <https://github.com/markjprice/apps-services-net10/blob/main/docs/ch04-blazor-libraries.md>.

Exercise 4.2 – Practice exercises

Create a Blazor component named `Carousel` that wraps the Bootstrap classes to work with carousels as a component, and then use it to show the eight categories in the Northwind database, including images.

You can learn about the Bootstrap carousel at the following link: <https://getbootstrap.com/docs/5.3/components/carousel/>.

Exercise 4.3 – Test your knowledge

Answer the following questions:

1. What is the benefit of the new Blazor Full Stack hosting model in .NET 8 compared to the legacy hosting models like Blazor Server?
2. Does Blazor WebAssembly support all features of the latest .NET APIs?
3. What is the file extension for Blazor components?
4. How do you set the default layout for all Blazor page components?
5. How do you register a route for a Blazor page component?
6. When would you set the `Match` property of a `<NavLink>` component

to `NavLinkMatch.All`?

7. You have imported a custom namespace in the `_Imports.razor` file, but when you try to use a class in that namespace in a code-behind file for the Blazor component, the class is not found. Why? How can you fix the issue?
8. What must you do to a property in a component class to have it set to a query string parameter automatically?
9. What is QuickGrid?
10. How can a Blazor component access browser features like local storage?

Exercise 4.4 – Explore topics

Use the links on the following page to learn more details about the topics covered in this chapter:

<https://github.com/markjprice/apps-services-net10/blob/main/docs/book-links.md#chapter-4---building-web-apps-using-blazor>

Summary

In this chapter, you learned:

- About some important concepts surrounding Blazor, like hosting models, components, routing, and how to pass parameters.
- How to build Blazor components with settable parameters, child content, and custom events.
- How to build Blazor components that get data from a database.

In the next chapter, you will learn how to implement popular third-party libraries.

Get This Book **UNLOCK NOW** **ion and Exclusive Extras**

Scan the QR code
book by name, co



Search for this
ps on the page.

Note: *Keep your invoice handy. Purchases made directly from Packt don't require one.*

5

Implementing Popular Third-Party Libraries

This chapter is about some popular third-party libraries for .NET that enable you to perform actions that either are not possible with the core .NET libraries or are better than the built-in functionality. These actions include manipulating images with **ImageSharp**, logging with **Serilog**, mapping objects to other objects with **AutoMapper**, making unit test assertions with **FluentAssertions**, validating data with **FluentValidation**, and generating PDFs with **QuestPDF**.

This chapter covers the following topics:

- Which third-party libraries are the most popular?
- Working with images
- Working with text and numbers
- Structured event data logging
- Mapping between objects
- Making fluent assertions in unit testing
- Validating data
- Generating PDFs

Which third-party libraries are the most popular?

To help me decide which third-party libraries to include in this book, I researched which are downloaded most frequently as of October 2025 at <https://www.nuget.org/stats/packages>, and, as shown in *Table 5.1*, they are:

Package	Downloads	Package	Downloads	Package	Downloads
Newtonsoft.Json	760,672,586	Microsoft.EntityFrameworkCore	345,657,253	Microsoft.AspNetCore.Mvc.Core	66,494,681
Microsoft.Extensions.DependencyInjection	55,947,579	Microsoft.AspNetCore.Mvc.Core	55,181,576	Microsoft.AspNetCore.Mvc.Core	47,831,946
Microsoft.AspNetCore.Mvc.Core	47,831,946	Microsoft.AspNetCore.Mvc.Core	46,638,955	Microsoft.AspNetCore.Mvc.Core	35,360,345
Microsoft.AspNetCore.Mvc.Core	35,360,345	Microsoft.AspNetCore.Mvc.Core	31,130,387	Microsoft.AspNetCore.Mvc.Core	40,365,740
Microsoft.AspNetCore.Mvc.Core	40,365,740	Microsoft.AspNetCore.Mvc.Core	19,708,692	Microsoft.AspNetCore.Mvc.Core	30,677,753
Microsoft.AspNetCore.Mvc.Core	30,677,753	Microsoft.AspNetCore.Mvc.Core	34,719,564	Microsoft.AspNetCore.Mvc.Core	30,520,726
Microsoft.AspNetCore.Mvc.Core	30,520,726	Microsoft.AspNetCore.Mvc.Core	37,241,502	Microsoft.AspNetCore.Mvc.Core	23,740,981
Microsoft.AspNetCore.Mvc.Core	23,740,981	Microsoft.AspNetCore.Mvc.Core	49,489,378	Microsoft.AspNetCore.Mvc.Core	69,957,836
Microsoft.AspNetCore.Mvc.Core	69,957,836	Microsoft.AspNetCore.Mvc.Core	79,815,116	Microsoft.AspNetCore.Mvc.Core	84,305,289
Microsoft.AspNetCore.Mvc.Core	84,305,289	Microsoft.AspNetCore.Mvc.Core		Microsoft.AspNetCore.Mvc.Core	

Table 5.1: The most downloaded NuGet packages

Warning! Third-party libraries change frequently,

including their license requirements. The inclusion of a third-party package in this book is not an endorsement or recommendation of that package, and especially not of future package versions.

What packages are covered in my books

My book *C# 14 and .NET 10 – Modern Cross-Platform Development Fundamentals* introduces:

- Processing JSON using `newtonsoft.json`
- Documenting web services using `Microsoft.AspNetCore.OpenApi` and `Scalar.AspNetCore` because `swashbuckle` is not maintained properly by its developer

Swashbuckle was popular for both generating the JSON document for a web service and for providing a user interface for trying it out because it was included by default in the ASP.NET Core Web API project template. In .NET 9, Microsoft removed Swashbuckle from the project templates due to a lack of maintenance. This is a good example of a package that still ranks highly for historical reasons even though you should avoid it in favor of more modern packages like `Scalar.AspNetCore`, as you will see in [Chapter 10, Building and Securing Minimal API Web Services](#).

Deploying to and integrating with **Amazon Web**

Services (AWS), and therefore all the `awssdk` packages, is out of scope for this book.

As well as raw download numbers, questions from readers and the usefulness of the library also contributed to my decision to include a library in this chapter, as summarized in the following list:

- Most popular library for manipulating images: **ImageSharp**
- Most popular library for manipulating text: **Humanizer**
- Most popular library for logging: **Serilog**
- Most popular library for object mapping: **AutoMapper**
- Most popular library for unit test assertions: **FluentAssertions**
- Most popular library for data validation: **FluentValidation**
- Open-source library for generating PDFs: **QuestPDF**

In [Chapter 6](#), *Handling Dates, Times, and Internationalization*, I cover the most popular library for handling dates and times: **Noda Time**.

In [Chapter 11](#), *Caching, Queuing, and Resilient Background Services*, I cover a few more popular libraries, as summarized in the following list:

- Most popular library for resilience and transient fault handling: **Polly**
- Most popular library for scheduling jobs and implementing background services: **Hangfire**
- Most popular library for distributed caching: **Redis**
- Most popular library for queuing: **RabbitMQ**

Now that you understand which packages I cover in this chapter and elsewhere in this book, let's get started with the first task that can be handled by a third-party package: working with images.

Working with images

ImageSharp is a third-party cross-platform 2D graphics library. When .NET Core 1.0 was in development, there was negative feedback from the community

about the missing `System.Drawing` namespace for working with 2D images. The ImageSharp project was started to fill that gap for modern .NET applications.

In their official documentation for `System.Drawing`, Microsoft says, “The `System.Drawing` namespace is not recommended for new development due to not being supported within a Windows or ASP.NET service, and it is not cross-platform. ImageSharp and SkiaSharp are recommended as alternatives.”

Six Labors released ImageSharp 3.0 in March 2023. It now requires .NET 6 or later and major future versions will target LTS releases of .NET, like .NET 10. You can read the announcement at the following link: <https://sixlabors.com/posts/announcing-imagesharp-300/>.

Generating grayscale thumbnails

Let’s see what can be achieved with ImageSharp:

1. Use your preferred code editor to create a console app project, as defined in the following list:
 - Project template: **Console App** / `console`
 - Solution file and folder: `PopularPackages`
 - Project file and folder: `WorkingWithImages`
 - **Do not use top-level statements**: Cleared
 - **Enable native AOT publish**: Cleared
7. In the `WorkingWithImages` project, create an `images` folder and download the nine images from the following link to it: <https://github.com/markjprice/apps-services-net10/tree/master/images/Categories>.
8. If you are using Visual Studio, then the `images` folder and its files must be copied to the `WorkingWithImages\bin\Debug\net10` folder

where the compiled console app will run. We can configure Visual Studio to do this for us, as shown in the following steps:

9. In **Solution Explorer**, select all nine images.
10. In **Properties**, set **Copy to Output Directory** to **Copy always**, as shown in *Figure 5.1*:

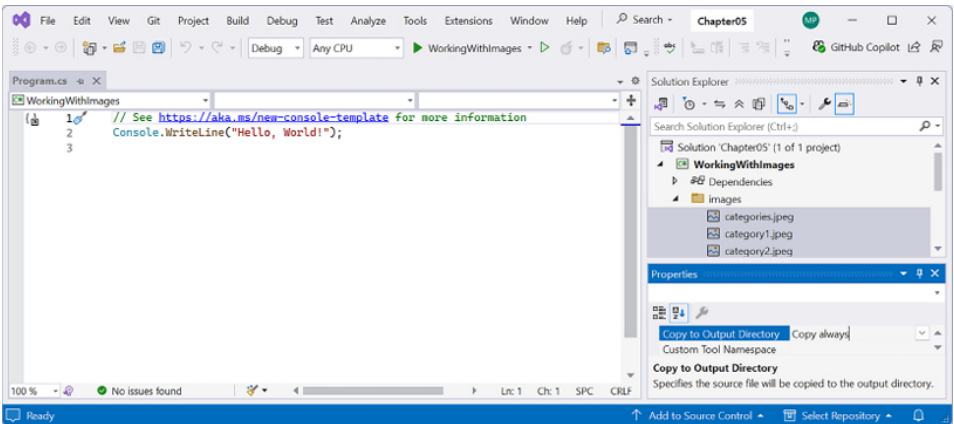


Figure 5.1: Setting images to always copy to the output directory

1. Open the project file and note the `<ItemGroup>` entries that will copy the nine images to the correct folder, as partially shown in the following markup:

```
<ItemGroup> <None Update="images
\categories.jpeg">
<CopyToOutputDirectory>Always</
CopyToOutputDirectory> </None> <None
Update="images\category1.jpeg">
<CopyToOutputDirectory>Always</
CopyToOutputDirectory> </None> ...
```

2. In the `WorkingWithImages` project, treat warnings as errors, globally and statically import the `System.Console` class, add a package reference for `SixLabors.ImageSharp`, and globally

import three ImageSharp namespaces that we will use types from, as shown highlighted in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup> <OutputType>Exe</
OutputType> <TargetFramework>net10.0</
TargetFramework> <ImplicitUsings>enable</
ImplicitUsings> <Nullable>enable</
Nullable> <TreatWarningsAsErrors>true</
TreatWarningsAsErrors> </PropertyGroup>
  <ItemGroup Label="Simplify use of the
Console class."> <Using
Include="System.Console" Static="true" />
</ItemGroup> <ItemGroup> <PackageReference
Include="SixLabors.ImageSharp" /> </
ItemGroup> <ItemGroup Label="Globally
import these namespaces."> <Using
Include="SixLabors.ImageSharp" /> <Using
Include="SixLabors.ImageSharp.PixelFormats"
/> <Using
Include="SixLabors.ImageSharp.Processing"
/> </ItemGroup> ...
```

To save space, in other steps like this in this chapter, I will not show the markup to treat warnings as errors or to globally and statically import the `System.Console` class. I will only show the `ItemGroup` and `PackageReference` for task-specific libraries.

1. Build the `WorkingWithImages` project.
2. In `Program.cs`, delete the existing statements and then add statements

to convert all the files in the `images` folder into grayscale thumbnails at one-tenth size, as shown in the following code:

```
string imagesFolder = Path.Combine(
    Environment.CurrentDirectory, "images");
WriteLine($"I will look for images in the
following folder:\n{imagesFolder}");
WriteLine(); if (!
Directory.Exists(imagesFolder)) {
WriteLine(); WriteLine("Folder does not
exist!"); return; } IEnumerable<string>
images =
Directory.EnumerateFiles(imagesFolder);
foreach (string imagePath in images) { if
(Path.GetFileNameWithoutExtension(imagePath).Ends
with("thumbnail")) { WriteLine($"Skipping:\n
{imagePath}"); WriteLine(); continue; //
This file has already been converted. }
string thumbnailPath = Path.Combine(
    Environment.CurrentDirectory, "images",
    Path.GetFileNameWithoutExtension(imagePath)
    + "-thumbnail" +
    Path.GetExtension(imagePath)); using
    (Image image = Image.Load(imagePath)) {
WriteLine($"Converting:\n {imagePath}");
WriteLine($"To:\n {thumbnailPath}");
image.Mutate(x => x.Resize(image.Width /
10, image.Height / 10)); image.Mutate(x =>
x.Grayscale()); image.Save(thumbnailPath);
WriteLine(); } } WriteLine("Image
processing complete."); if
(OperatingSystem.IsWindows()) {
Process.Start("explorer.exe",
imagesFolder); } else { WriteLine("View
the images folder."); }
```

3. Run the console app and note the images should be converted into grayscale thumbnails, as shown in the following partial output:

```
I will look for images in the following
folder: C:\apps-services-
net10\PopularPackages\WorkingWithImages
\bin\Debug\net10.0\images Converting: C:
\apps-services-net10\PopularPackages
\WorkingWithImages\bin\Debug
\net10.0\images\categories.jpeg To: C:
\apps-services-net10\PopularPackages
\WorkingWithImages\bin\Debug
\net10.0\images\categories-thumbnail.jpeg
Converting: C:\apps-services-
net10\PopularPackages\WorkingWithImages
\bin\Debug\net10.0\images\category1.jpeg
To: C:\apps-services-net10\PopularPackages
\WorkingWithImages\bin\Debug
\net10.0\images\category1-thumbnail.jpeg
... Converting: C:\apps-services-
net10\PopularPackages\WorkingWithImages
\bin\Debug\net10.0\images\category8.jpeg
To: C:\apps-services-net10\PopularPackages
\WorkingWithImages\bin\Debug
\net10.0\images\category8-thumbnail.jpeg
Image processing complete.
```

4. In the filesystem, open the appropriate `images` folder and note the much-smaller-in-bytes grayscale thumbnails, as shown in *Figure 5.2*:

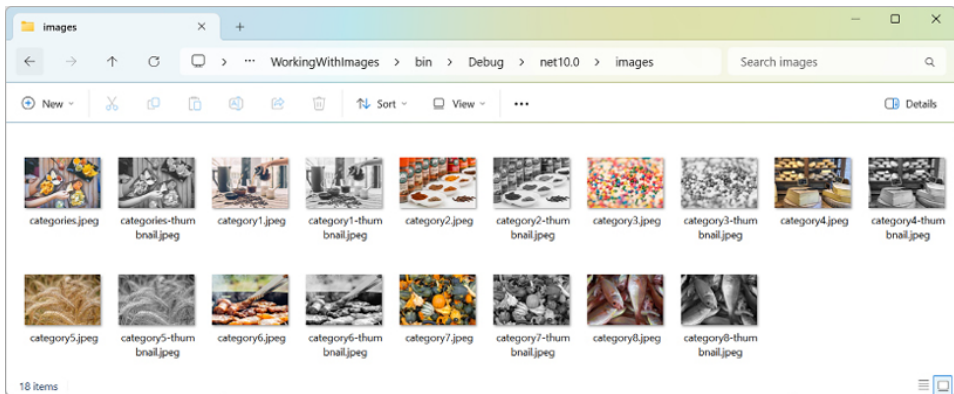


Figure 5.2: Images after processing

ImageSharp packages for drawing and the web

ImageSharp also has NuGet packages for programmatically drawing images and working with images on the web, as shown in the following list:

- `SixLabors.ImageSharp.Drawing`
- `SixLabors.ImageSharp.Web`

Learn more details at the following link:

<https://docs.sixlabors.com/>.

Working with text and numbers

Humanizer manipulates text in `string` values, names of `enum` values, dates, times, numbers, and quantities.

Working with text

The built-in `string` type has methods to manipulate text like `Substring` and `Trim`. But there are plenty of other common manipulations that we might

want to perform on text. For example:

- You might have an ugly string generated by some code and you want to make it look friendlier to display to a user. This is common with `enum` types that cannot use spaces in multi-word values.
- You might build a web content management system, and when a user enters an article title, you need to convert what they enter into a format suitable for a URL path.
- You might have a long `string` value that needs to be truncated to show in the limited space of a mobile user interface.

In the next three subsections, you will review tables of text transformations and methods. After that, you will write some code to show example text transformations.

Humanizer case transformations

Complex transformations can be performed in sequence by passing multiple Humanizer transformations to the `Transform` method. Transforms implement the `IStringTransformer` or `ICulturedStringTransformer` interfaces so you can implement your own custom transforms.

The built-in transforms are all casing transformations, and they are listed in *Table 5.2* along with convenient alternative methods that extend the `string` type:

Description				
Transform all characters in the <code>string</code> to lowercase				
Transform all characters in the <code>string</code> to uppercase				
Transform the first character of each word in the <code>string</code> to uppercase				
Transform the first character of each word in the <code>string</code> to uppercase. It ignores periods (full stops) so it does not recognize sentences.				

Table 5.2: Humanizer casing transforms

Good practice: It is important to consider the casing of the original text. If it is already uppercase, the title and sentence casing options will *not* convert to lowercase! You might need to transform to lowercase first, then transform to title or sentence case.

As well as calling the `Transform` method with a transform object like `To.TitleCase`, there are convenience methods for manipulating the case of text, as shown in *Table 5.3*:

Using extension method		
Equivalent to transforming with <code>To.TitleCase</code>		
Converts strings to upper camel case, also removing underscores		
Same as <code>ToLower</code> except that the first character is lowercase		

Table 5.3: Humanizer text extension methods

Humanizer spacing conversions

There are convenient methods for manipulating the spacing of text, by adding underscores and dashes, as shown in *Table 5.4*:

Using extension method		
Separates the input words with an underscore		
Replaces underscores with dashes (hyphens) in the string		
Separates the input words with dashes (hyphens)		

Table 5.4: Humanizer spacing conversion methods

Humanizer’s Singularize and Pluralize methods

Humanizer has two extension methods for the `string` class that automate

converting between the singular and plural forms of a word, as shown in *Table 5.5*:

Singularization method		
If the starting contains a plural word, it is converted to the singular equivalent.		
If the starting contains a singular word, it is converted to the plural equivalent.		

Table 5.5: Humanizer’s Singularize and Pluralize methods

These methods are used by Microsoft Entity Framework Core to singularize and pluralize the names of entity classes and their members.

Exploring text manipulations with a console app

Let’s explore some examples of text manipulation using Humanizer:

1. Use your preferred code editor to add a new **Console App** / `console` project named `HumanizingData` to the `PopularPackages` solution.
2. In the `HumanizingData` project, treat warnings as errors, globally and statically import the `System.Console` class, and add a package reference for `Humanizer`, as shown in the following markup:

```
<ItemGroup> <PackageReference  
  Include="Humanizer" /> </ItemGroup>
```

We are referencing a `Humanizer` package that includes all languages. If you only need the English language, then you can reference

Humanizer.Core instead. If you also need a subset of languages, reference specific language packages using the pattern Humanizer.Core.<lang>, for example, Humanizer.Core.fr for French.

1. Build the HumanizingData project to restore packages.
2. Add a new class file to the project named Program.Functions.cs.
3. In Program.Functions.cs, add statements to import the namespace for working with globalization, and to define a method to configure the console to enable easy switching of the current culture and enable the use of special characters, as shown in the following code:

```
using System.Globalization; // To use
CultureInfo. partial class Program {
    private static void
    ConfigureConsole(string culture = "en-US")
    { // To enable special characters like ...
      (ellipsis) as a single character.
      OutputEncoding =
      System.Text.Encoding.UTF8; Thread t =
      Thread.CurrentThread; t.CurrentCulture =
      CultureInfo.GetCultureInfo(culture);
      t.CurrentUICulture = t.CurrentCulture;
      WriteLine("Current culture: {0}",
      t.CurrentCulture.DisplayName);
      WriteLine(); } }
```

4. In Program.cs, delete the existing statements, and then call the ConfigureConsole method, as shown in the following code:

```
ConfigureConsole(); // Defaults to en-US
culture.
```

5. In `Program.Functions.cs`, add a statement to import the namespace for working with extension methods provided by `Humanizer`, as shown in the following code:

```
using Humanizer; // To use common
Humanizer extension methods.
```

6. In `Program.Functions.cs`, add statements to define a method to output an original string and then the results of transforming it using the built-in casing transforms, as shown in the following code:

```
private static void OutputCasings(string
original) { WriteLine("Original casing:
{0}", original); WriteLine("Lower casing:
{0}", original.Transform(To.LowerCase));
WriteLine("Upper casing: {0}",
original.Transform(To.UpperCase));
WriteLine("Title casing: {0}",
original.Transform(To.TitleCase));
WriteLine("Sentence casing: {0}",
original.Transform(To.SentenceCase));
WriteLine("Lower, then Sentence casing:
{0}", original.Transform(To.LowerCase,
To.SentenceCase)); WriteLine(); }
```

7. In `Program.cs`, call the `OutputCasings` method with three different string values, as shown in the following code:

```
OutputCasings("The cat sat on the mat.");
OutputCasings("THE CAT SAT ON THE MAT.");
OutputCasings("the cat sat on the mat. the
```

```
frog jumped.");
```

8. Run the code and view the result, as shown in the following output:

```
Current culture: English (United States)
Original casing: The cat sat on the mat.
Lower casing: the cat sat on the mat.
Upper casing: THE CAT SAT ON THE MAT.
Title casing: The Cat Sat on the Mat.
Sentence casing: The cat sat on the mat.
Lower, then Sentence casing: The cat sat
on the mat. Original casing: THE CAT SAT
ON THE MAT. Lower casing: the cat sat on
the mat. Upper casing: THE CAT SAT ON THE
MAT. Title casing: THE CAT SAT ON THE MAT.
Sentence casing: THE CAT SAT ON THE MAT.
Lower, then Sentence casing: The cat sat
on the mat. Original casing: the cat sat
on the mat. the frog jumped. Lower casing:
the cat sat on the mat. the frog jumped.
Upper casing: THE CAT SAT ON THE MAT. THE
FROG JUMPED. Title casing: The Cat Sat on
the Mat. the Frog Jumped. Sentence casing:
The cat sat on the mat. the frog jumped.
Lower, then Sentence casing: The cat sat
on the mat. the frog jumped.
```

9. In `Program.Functions.cs`, add statements to define a method that outputs an ugly string value using various `Humanizer` extension methods, as shown in the following code:

```
private static void
OutputSpacingAndDashes() { string ugly =
```

```

"ERROR_MESSAGE_FROM_SERVICE";
WriteLine("Original string: {0}", ugly);
WriteLine("Humanized: {0}",
ugly.Humanize()); // LetterCasing is
legacy and will be removed in future.
WriteLine("Humanized, lower case: {0}",
ugly.Humanize(LetterCasing.LowerCase)); //
Use Transform for casing instead.
WriteLine("Transformed (lower case, then
sentence case): {0}",
ugly.Transform(To.LowerCase,
To.SentenceCase)); WriteLine("Humanized,
Transformed (lower case, then sentence
case): {0}",
ugly.Humanize().Transform(To.LowerCase,
To.SentenceCase)); }

```

10. In `Program.cs`, comment out the previous method calls and then add a statement to call the new method, as shown highlighted in the following code:

```

/* OutputCasings("The cat sat on the
mat."); OutputCasings("THE CAT SAT ON THE
MAT."); OutputCasings("the cat sat on the
mat. the frog jumped."); */
OutputSpacingAndDashes ();

```

11. Run the code and view the result, as shown in the following output:

```

Original string:
ERROR_MESSAGE_FROM_SERVICE Humanized:
ERROR MESSAGE FROM SERVICE Humanized,
lower case: error message from service

```

```
Transformed (lower case, then sentence  
case): Error_message_from_service  
Humanized, Transformed (lower case, then  
sentence case): Error message from service
```

12. Add a new class file to the project named WondersOfTheAncientWorld.cs.
13. Modify the WondersOfTheAncientWorld.cs file, as shown in the following code:

```
namespace Packt.Shared; public enum  
WondersOfTheAncientWorld : byte { None =  
0b_0000_0000, // i.e. 0 GreatPyramidOfGiza  
= 0b_0000_0001, // i.e. 1  
HangingGardensOfBabylon = 0b_0000_0010, //  
i.e. 2 StatueOfZeusAtOlympia =  
0b_0000_0100, // i.e. 4  
TempleOfArtemisAtEphesus = 0b_0000_1000,  
// i.e. 8 MausoleumAtHalicarnassus =  
0b_0001_0000, // i.e. 16 ColossusOfRhodes  
= 0b_0010_0000, // i.e. 32  
LighthouseOfAlexandria = 0b_0100_0000 //  
i.e. 64 }
```

14. In Program.Functions.cs, import the namespace to use the enum that we just defined, as shown in the following code:

```
using Packt.Shared; // To use  
WondersOfTheAncientWorld.
```

15. In Program.Functions.cs, define a method to create the WondersOfTheWorld variable and output its name using various

Humanizer extension methods, as shown in the following code:

```
private static void OutputEnumNames() {  
    var favoriteAncientWonder =  
        WondersOfTheAncientWorld.StatueOfZeusAtOlympia;  
    WriteLine("Raw enum value name: {0}",  
        favoriteAncientWonder);  
    WriteLine("Humanized: {0}",  
        favoriteAncientWonder.Humanize());  
    WriteLine("Humanized, then Titleized:  
        {0}",  
        favoriteAncientWonder.Humanize().Titleize());  
    WriteLine("Truncated to 8 characters:  
        {0}",  
        favoriteAncientWonder.ToString().Truncate(length:  
            8));  
    WriteLine("Kebabized: {0}",  
        favoriteAncientWonder.ToString().Kebabize());  
}
```

16. In `Program.cs`, comment out the previous method call, and then add a call to `OutputEnumNames`, as shown highlighted in the following code:

```
//OutputSpacingAndDashes();  
OutputEnumNames();
```

17. Run the code and view the result, as shown in the following output:

```
Raw enum value name: StatueOfZeusAtOlympia  
Humanized: Statue of zeus at olympia  
Humanized, then Titlerized: Statue of Zeus  
at Olympia Truncated to 8 characters:  
StatueO... Kebabized: statue-of-zeus-at-
```


Note the `Truncate` method uses the single-character `...` (ellipsis) by default. If you ask to truncate to a length of 8, it can return the first seven characters followed by the ellipsis character. You can specify a different character using an overload of the `Truncate` method.

Working with numbers

Now let's see how `Humanizer` can help us with numbers:

1. In `Program.Functions.cs`, define a method to create some numbers and then output them using various `Humanizer` extension methods, as shown in the following code:

```
private static void NumberFormatting() {
    int number = 123; WriteLine($"Original
number: {number}"); WriteLine($"Roman:
{number.ToRoman()}"); WriteLine($"Words:
{number.ToWords()}"); WriteLine($"Ordinal
words: {number.ToOrdinalWords()}");
    WriteLine(); string[] things = { "fox",
    "person", "sheep", "apple", "goose",
    "oasis", "potato", "die", "dwarf",
    "attorney general", "biceps"}; for (int i
    = 1; i <= 3; i++) { for (int j = 0; j <
    things.Length; j++) {
        Write(things[j].ToQuantity(i,
        ShowQuantityAs.Words)); if (j <
        things.Length - 1) Write(", "); }
    WriteLine(); } WriteLine(); int thousands
```

```

= 12345; int millions = 123456789;
WriteLine("Original: {0}, Metric: About
{1}", thousands,
thousands.ToMetric(decimals: 0));
WriteLine("Original: {0}, Metric: {1}",
thousands,
thousands.ToMetric(MetricNumeralFormats.WithSpace
| MetricNumeralFormats.UseShortScaleWord,
decimals: 0)); WriteLine("Original: {0},
Metric: {1}", millions,
millions.ToMetric(decimals: 1)); }

```

2. In `Program.cs`, comment out the previous method call and add a call to `NumberFormatting`, as shown highlighted in the following code:

```

//OutputEnumNames(); NumberFormatting();

```

3. Run the code and view the result, as shown in the following output:

```

Original number: 123 Roman: CXXIII Words:
one hundred and twenty-three Ordinal
words: hundred and twenty-third one fox,
one person, one sheep, one apple, one
goose, one oasis, one potato, one die, one
dwarf, one attorney general, one bicep two
foxes, two people, two sheep, two apples,
two geese, two oases, two potatoes, two
dice, two dwarves, two attorney generals,
two biceps three foxes, three people,
three sheep, three apples, three geese,
three oases, three potatoes, three dice,
three dwarves, three attorney generals,
three biceps Original: 12345, Metric:

```

About 12k Original: 12345, Metric: About
12 thousand Original: 123456789, Metric:
123.5M

Humanizer's default vocabulary is quite decent, but it does not correctly pluralize attorney general (the plural is *attorneys general*) or biceps (the singular is *biceps* and the plural is *bicepses*).

1. In `Program.Functions.cs`, at the top of the `NumberFormatting` method, add statements to register two irregular words, as shown in the following code:

```
Vocabularies.Default.AddIrregular("biceps",  
    "bicepses");  
Vocabularies.Default.AddIrregular("attorney  
    general", "attorneys general");
```

2. Run the code, view the result, and note that the two irregular words now output correctly, as shown in the following output:

```
one fox, one person, one sheep, one apple,  
one goose, one oasis, one potato, one die,  
one dwarf, one attorney general, one  
biceps two foxes, two people, two sheep,  
two apples, two geese, two oases, two  
potatoes, two dice, two dwarves, two  
attorneys general, two bicepses three  
foxes, three people, three sheep, three  
apples, three geese, three oases, three  
potatoes, three dice, three dwarves, three
```

Working with dates and times

Now let's see how Humanizer can help us with dates and times:

1. In `Program.Functions.cs`, define a method to get the current date and time and some number of days, and then output them using various Humanizer extension methods, as shown in the following code:

```
private static void DateTimeFormatting() {
    DateTimeOffset now = DateTimeOffset.Now;
    // By default, all Humanizer comparisons
    are to Now (UTC). WriteLine($"Now (UTC):
    {now}"); WriteLine("Add 3 hours,
    Humanized: {0}",
    now.AddHours(3).Humanize());
    WriteLine("Add 3 hours and 1 minute,
    Humanized: {0}",
    now.AddHours(3).AddMinutes(1).Humanize());
    WriteLine("Subtract 3 hours, Humanized:
    {0}", now.AddHours(-3).Humanize());
    WriteLine("Add 24 hours, Humanized: {0}",
    now.AddHours(24).Humanize());
    WriteLine("Add 25 hours, Humanized: {0}",
    now.AddHours(25).Humanize());
    WriteLine("Add 7 days, Humanized: {0}",
    now.AddDays(7).Humanize()); WriteLine("Add
    7 days and 1 minute, Humanized: {0}",
    now.AddDays(7).AddMinutes(1).Humanize());
    WriteLine("Add 1 month, Humanized: {0}",
    now.AddMonths(1).Humanize()); WriteLine();
    // Examples of TimeSpan humanization.
    int[] daysArray = { 12, 13, 14, 15, 16 };
```

```

foreach (int days in daysArray) {
    WriteLine("{0} days, Humanized: {1}",
        days, TimeSpan.FromDays(days).Humanize());
    WriteLine("{0} days, Humanized with
precision 2: {1}", days,
        TimeSpan.FromDays(days).Humanize(precision:
2)); WriteLine("{0} days, Humanized with
max unit days: {1}", days,
        TimeSpan.FromDays(days).Humanize( maxUnit:
TimeUnit.Day)); WriteLine(); } // Examples
of clock notation. TimeOnly[] times = {
    new TimeOnly(9, 0), new TimeOnly(9, 15),
    new TimeOnly(15, 30) }; foreach (TimeOnly
time in times) { WriteLine("{0}: {1}",
    time, time.ToClockNotation()); } }

```

2. In Program.cs, comment out the previous method call and add a call to `DateTimeFormatting`, as shown highlighted in the following code:

```

//NumberFormatting();
DateTimeFormatting();

```

3. Run the code and view the result, as shown in the following output:

```

Current culture: English (United States)
Now (UTC): 5/30/2023 8:12:51 AM +01:00 Add
3 hours, Humanized: 2 hours from now Add 3
hours and 1 minute, Humanized: 3 hours
from now Subtract 3 hours, Humanized: 3
hours ago Add 24 hours, Humanized: 23
hours from now Add 25 hours, Humanized:
tomorrow Add 7 days, Humanized: 6 days

```

```
from now Add 7 days and 1 minute,  
Humanized: 7 days from now Add 1 month,  
Humanized: one month from now 12 days,  
Humanized: 1 week 12 days, Humanized with  
precision 2: 1 week, 5 days 12 days,  
Humanized with max unit days: 12 days 13  
days, Humanized: 1 week 13 days, Humanized  
with precision 2: 1 week, 6 days 13 days,  
Humanized with max unit days: 13 days 14  
days, Humanized: 2 weeks 14 days,  
Humanized with precision 2: 2 weeks 14  
days, Humanized with max unit days: 14  
days 15 days, Humanized: 2 weeks 15 days,  
Humanized with precision 2: 2 weeks, 1 day  
15 days, Humanized with max unit days: 15  
days 16 days, Humanized: 2 weeks 16 days,  
Humanized with precision 2: 2 weeks, 2  
days 16 days, Humanized with max unit  
days: 16 days 9:00 AM: nine o'clock 9:15  
AM: a quarter past nine 3:30 PM: half past  
three
```

4. In `Program.cs`, specify the French language and region when configuring the console, as shown highlighted in the following code:

```
ConfigureConsole("fr-FR"); // Defaults to  
en-US culture.
```

5. Run the code, view the result, and note that the text is localized to French.

You can learn more about Humanizer's capabilities at the following link: <https://github.com/Humanizr/Humanizer>.

Now let's move on to logging, which is important because it is a foundational practice that directly affects maintainability, reliability, and your ability to diagnose and fix issues quickly.

Structured event data logging

The most obvious reason for logging is to understand what went wrong and when. In .NET, exceptions can be caught, and stack traces can be printed, but that's often not enough context. Although .NET includes logging frameworks, third-party logging providers give more power and flexibility by using **structured event data**. Serilog is the most popular.

Structured event data

Most systems write plain text messages to their logs.

Serilog can be told to write serialized structured data to the log. The `@` symbol prefixing a parameter tells Serilog to serialize the object passed in, instead of just the result of calling the `ToString` method.

Later, that complex object can be queried for improved search and sort capabilities in the logs.

For example:

```
var lineitem = new { ProductId = 11, UnitPrice
= 25.49, Quantity = 3 };
log.Information("Added {@LineItem} to shopping
cart.", lineitem);
```

You can learn more about how Serilog handles structured data at the following link: <https://github.com/serilog/serilog/wiki/Structured-Data>.

Serilog sinks

All logging systems need to record the log entries somewhere. That could be to the console output, a file, or a more complex data store like a relational database or cloud data store. Serilog calls these **sinks**.

Serilog has hundreds of official and third-party sink packages for all the possible places you might want to record your logs. To use them, just include the appropriate package. The most popular are shown in the following list:

- `serilog.sinks.file`: Writes logs to a specified file path on the local file system or network share. You provide the file name (absolute or relative). You can configure rolling behavior like daily files using parameters like `rollingInterval: RollingInterval.Day`. The file is written synchronously or asynchronously, depending on the configuration. By default, logs are written in plain text, but JSON formatting can be enabled.
- `serilog.sinks.console`: Writes logs to the standard output stream (`stdout`), typically visible in the terminal/console if you're running locally, container logs such as Docker or Kubernetes, or cloud platform log collectors such as Azure App Service or AWS CloudWatch. This is good for development and containerized environments. It can output color-coded text for log levels when using themes such as `SystemConsoleTheme.Literate`.
- `serilog.sinks.async`: This sink doesn't have a destination of its own because it's an infrastructure sink that wraps another sink to improve performance. `WriteTo.Async(...)` is an official Serilog wrapper that queues log events in memory and ships them to the wrapped sink on a background thread. It's ideal for sinks like File and Console that can block on I/O. You can cap the in-memory queue or choose to block when it's full, as shown in the following code:

```
.WriteTo.Async(a => a.File("log-.txt",
```



```
rollingInterval: RollingInterval.Day),  
bufferSize: 5000, blockWhenFull: false)  
.WriteTo.Async(a => a.Console(),  
bufferSize: 5000)
```

- `serilog.sinks.periodicbatching`: This sink doesn't have a destination of its own because it's an infrastructure sink that wraps another sink to improve performance. It collects log events in memory and then periodically flushes them as a batch to another sink. It is used internally by other sinks to reduce I/O overhead.
- `serilog.sinks.debug`: Writes logs to the debug output window, visible in Visual Studio's Output pane (when "Show output from: Debug" is selected), and any debugger attached to the process such as Rider or VS Code Debug Console.
- `serilog.sinks.applicationinsights`: Writes logs to your Azure Application Insights resource, visible in the Logs (Analytics) or Traces section of the Azure Portal, via the Application Insights Telemetry API.
- `serilog.sinks.mssqlserver`: Writes logs to a table in a SQL Server database. This is good for enterprise applications where logs need to be queryable via SQL or integrated with BI tools.

Good practice: If you're building a modern .NET app or service, the most common setup is a hybrid of console (for container logs), file (for on-disk persistence), and a cloud log like Application Insights (for centralized monitoring).

There are more than 470 packages currently listed on Microsoft's public NuGet feed: <https://www.nuget.org/packages?q=serilog.sinks>.

Logging to the console and a rolling file with Serilog

Let's start:

1. Use your preferred code editor to add a new **Console App** / console project named Serilogging to a PopularPackages solution.
2. In the Serilogging project, treat warnings as errors, globally and statically import the `System.Console` class, and add a package reference for Serilog, including sinks for console, async, and file (which also supports rolling files), as shown in the following markup:

```
<ItemGroup> <PackageReference
Include="Serilog" /> <PackageReference
Include="Serilog.Sinks.Console" />
<PackageReference
Include="Serilog.Sinks.File" />
<PackageReference
Include="Serilog.Sinks.Async" /> </
ItemGroup>
```

3. Build the Serilogging project.
4. In the Serilogging project, add a new folder named Models.
5. In the Serilogging project, in the Models folder, add a new class file named `ProductPageView.cs`, and modify its contents, as shown in the following code:

```
namespace Serilogging.Models; public class
ProductPageView { public int ProductId {
get; set; } public string? PageTitle {
get; set; } public string? SiteSection {
get; set; } }
```

6. In `Program.cs`, delete the existing statements and then import some namespaces for working with Serilog, as shown in the following code:

```
using Serilog; // To use Log,
               // LoggerConfiguration, RollingInterval.
using Serilog.Core; // To use Logger.
using Serilogging.Models; // To use
               // ProductPageView.
```

7. In `Program.cs`, create a logger configuration that writes to the console and configures a rolling interval so a new file is created each day, and write various levels of log entries, as shown in the following code:

```
// Create a new logger that will write to
// the console and to // a text file, one-
// file-per-day, named with the date. using
// Logger log = new LoggerConfiguration() //
// console and file both executed on a
// background worker. .WriteTo.Async(a =>
// a.Console()) .WriteTo.Async(a =>
// a.File("log.txt", rollingInterval:
// RollingInterval.Day)) .CreateLogger(); //
// Assign the new logger to the static entry
// point for logging. Log.Logger = log;
Log.Information("The global logger has
been configured."); // Log some example
// entries of differing severity.
Log.Warning("Danger, Serilog, danger!");
Log.Error("This is an error!");
Log.Fatal("Fatal problem!");
ProductPageView pageView = new() {
    PageTitle = "Chai", SiteSection =
```

```
"Beverages", ProductId = 1 };
Log.Information("{@PageView} occurred at
{Viewed}", pageView,
DateTimeOffset.UtcNow); // For a log with
a buffer, like a text file logger, you //
must flush before ending the app.
Log.CloseAndFlush();
```

8. Run the console app and note the messages, as shown in the following output:

```
[07:09:43 INF] The global logger has been
configured. [07:09:43 WRN] Danger,
Serilog, danger! [07:09:43 ERR] This is an
error! [07:09:43 FTL] Fatal problem!
[07:09:43 INF] {"ProductId": 1,
"PageTitle": "Chai", "SiteSection":
"Beverages", "$type": "ProductPageView"}
occurred at 09/07/2023 15:08:44 +00:00
```

9. Open the `logYYYYMMDD.txt` file, where `YYYY` is the year, `MM` is the month, and `DD` is the day, and note it contains the same messages:

- For Visual Studio, the log file will be written to the `Serilogging\bin\Debug\net10.0` folder.
- For VS Code and `dotnet run`, the log file will be written to the `Serilogging` folder.

Learn more details at the following link:

<https://serilog.net/>.

Good practice: Disable logging when it is not needed. Logging can get costly fast. For example,

one organization was spending \$10,000 per month on cloud resources, much more than predicted, and they didn't know why. It turned out they were logging every SQL statement executed in production. By “flipping a switch” to stop that logging, they saved \$8,500 per month! You can read Milan Jovanović's story at the following link: https://www.linkedin.com/posts/milan-jovanovic_i-helped-a-team-save-100k-in-azure-cloud-activity-7109887614664474625-YDiU/.

Avoid logging sensitive data

`Serilog.Enrichers.Sensitive` is a community package (`serilog-contrib`) whose goal is to help you mask sensitive/PII data automatically in your Serilog log events so that you don't inadvertently leak things like email addresses, account numbers, and so on in your logging streams. It acts as a log event filter that intercepts log messages and masks sensitive values in both message templates and the properties attached to the log event.

By default, it supports masking email addresses and IBAN values (International Bank Account Numbers) out of the box.

You can configure it in two modes:

1. **Always mask**: treats all occurrences of sensitive patterns such as email and IBAN in message templates or properties.
2. **Mask only in “sensitive areas”**: only masks when you annotate or mark certain contexts as sensitive.

It also supports customizing the mask string. By default, it uses `***MASKED***`. You can list property names to be masked explicitly via its options such as `MaskProperties`, `MaskValue`, or even custom masking operators.

Let's see some practical code examples so you can better understand common masking operations.

When a log line contains something like the following:

```
logger.Information("This is a sensitive  
{Email}", "james.bond@domain.com");
```

It will output something like: This is a sensitive *****MASKED*****, rather than the actual email address.

So effectively it is a sanitizer or redactor layer for Serilog.

Here's a minimal code example:

```
Log.Logger = new LoggerConfiguration()  
    .Enrich.WithSensitiveDataMasking() // add the  
enricher .WriteTo.Console()  
    .WriteTo.File("logs/log-.txt",  
        rollingInterval: RollingInterval.Day)  
    .CreateLogger(); Log.Information("User logged  
in with email {Email}", "user@example.com");
```

That will suppress/log the masked version instead of the raw email.

To summarize, let's review some recommendations:

- Use `Serilog.Enrichers.Sensitive` (or a similar masking layer) as a defensive layer, not your only layer. You shouldn't rely on it to catch all sensitive logging; rational design and discipline in what you log still matter most.
- Explicitly identify the kinds of data you deem sensitive such as emails, SSNs, tokens, and so on, and then write a custom masking operator if needed.
- Test the integration end-to-end with your sinks (especially remote ones)

like Application Insights) to ensure the masking really gets applied before data leaves your process.

- Keep your mask configuration versioned and track new sensitive fields so your masking policy evolves along with your data model.

You can learn more about this package at the following link: <https://github.com/serilog-contrib/Serilog.Enrichers.Sensitive>.

Now let's review your options when you need to map between objects.

Mapping between objects

One of the most boring parts of being a programmer is mapping between objects. It is common to need to integrate systems or components that have conceptually similar objects but with different structures.

Models for data are different for different parts of an application. Models that represent data in storage are often called **entity models**. Models that represent data that must be passed between layers are often called **data transfer objects (DTOs)**. Models that represent only the data that must be presented to a user are often called **view models**. All these models are likely to have commonalities but different structures.

Introducing AutoMapper

AutoMapper is a popular package for mapping objects because it has conventions that make the work as easy as possible. For example, if you have a source member called `CompanyName`, it will be mapped to a destination member with the name `CompanyName`.

AutoMapper's creator, Jimmy Bogard, has written

an article about its design philosophy that is worth reading, available at the following link: <https://jimmybogard.com/automappers-design-philosophy/>. Also be aware that Jimmy's OSS projects have gone commercial: <https://www.jimmybogard.com/automapper-and-mediator-going-commercial/>. We will use AutoMapper 14 to avoid the commercial license required by AutoMapper 15 and later. Alternatives include Mediator: <https://github.com/martinothamar/Mediator> and Mapperly: <https://github.com/riok/mapperly>.

Let's see an example of AutoMapper in action. You will create four projects:

- A class library for the entity and view models.
- A class library to create mapper configurations for reuse in unit tests and actual projects.
- A unit test project to test the mappings.
- A console app to perform a live mapping.

We will construct an example object model that represents an e-commerce website customer and their shopping cart with a couple of items, and then map it to a summary view model to present to the user.

Defining models for an AutoMapper configuration

To test the mapping, we will define some `record` types. As a reminder, a `record` (or `record class`) is a reference type that has value-based equality. A `class` is a reference type that has memory address-based equality

(except for `string`, which overrides this behavior).

It is good practice to always validate your configuration for mappings before using them, so we will start by defining some models and a mapping between them, and then create a unit test for the mappings:

1. Use your preferred code editor to add a new **Class Library** / `classlib` project named `MappingObjects.Models` to the `PopularPackages` solution.
2. In the `MappingObjects.Models` project, delete the file named `Class1.cs`.
3. In the `MappingObjects.Models` project, add a new class file named `Customer.cs` and modify its contents to define an immutable record type named `Customer` using constructor syntax, as shown in the following code:

```
namespace Northwind.EntityModels; // This
record will only have a constructor with
the parameters below. // Objects will be
immutable after instantiation using this
constructor. // It will not have a default
parameterless constructor. public record
class Customer( string FirstName, string
LastName );
```

4. In the `MappingObjects.Models` project, add a new class file named `LineItem.cs` and modify its contents, as shown in the following code:

```
namespace Northwind.EntityModels; public
record class LineItem( string ProductName,
decimal UnitPrice, int Quantity );
```

5. In the `MappingObjects.Models` project, add a new class file named `Cart.cs` and modify its contents, as shown in the following code:

```
namespace Northwind.EntityModels; public
record class Cart( Customer Customer,
List<LineItem> Items );
```

6. In the `MappingObjects.Models` project, add a new class file named `Summary.cs` in the `Northwind.ViewModels` namespace (not `Northwind.EntityModels`), delete any existing statements, and then define a record type that can have its properties set *after* instantiating it with a default parameterless constructor, as shown in the following code:

```
namespace Northwind.ViewModels; public
record class Summary { // These properties
can be initialized once but then never
changed. public string? FullName { get;
init; } public decimal Total { get; init;
} // This record class will have a default
parameterless constructor. // The
following commented statement is
automatically generated: // public
Summary() { } }
```

For the entity models, we used `record class` types defined using the constructor syntax to make them immutable. But an instance of `Summary` will need to be created using a default parameterless constructor, and then its members set by

AutoMapper. Therefore, it must be a `record` class with public properties that can be set during initialization, but not after that. To do this, we use the `init` keyword.

Defining mappers for an AutoMapper configuration

Now we can define the mappings between models:

1. Use your preferred code editor to add a new **Class Library** / `classlib` project named `MappingObjects.Mappers` to the `PopularPackages` solution.
2. In the `MappingObjects.Mappers` project, treat warnings as errors, add a reference to the latest `AutoMapper` package, and add a reference to the `Models` project, as shown in the following markup:

```
<ItemGroup> <PackageReference  
  Include="AutoMapper" /> </ItemGroup>  
<ItemGroup> <ProjectReference Include= "..  
  \MappingObjects.Models  
  \MappingObjects.Models.csproj" /> </  
ItemGroup>
```

3. Build the `MappingObjects.Mappers` project to restore packages and compile referenced projects.
4. In the `MappingObjects.Mappers` project, delete the file named `Class1.cs`.
5. In the `MappingObjects.Mappers` project, add a new class file named `CartToSummaryMapper.cs` and modify its contents to create a mapper configuration that maps the `FullName` of the `Summary` to a combination of the `FirstName` and `LastName` from

Customer, as shown in the following code:

```
using AutoMapper; // To use
MapperConfiguration. using
AutoMapper.Internal; // To use the
Internal() extension method. using
Northwind.EntityModels; // To use Cart.
using Northwind.ViewModels; // To use
Summary. namespace MappingObjects.Mappers;
public static class CartToSummaryMapper {
public static MapperConfiguration
GetMapperConfiguration() {
MapperConfiguration config = new(cfg => {
// To fix an issue with the MaxInteger
method: // https://github.com/AutoMapper/
AutoMapper/issues/3988
cfg.Internal().MethodMappingEnabled =
false; // Configure the mapper using
projections. cfg.CreateMap<Cart,
Summary>() // Map the first and last names
formatted to the full name.
.ForMember(dest => dest.FullName, opt =>
opt.MapFrom(src => string.Format("{0}
{1}", src.Customer.FirstName,
src.Customer.LastName) )); }); return
config; } }
```

Performing tests for an AutoMapper configuration

Now we can define unit tests for the mapper:

1. Use your preferred code editor to add a new **xUnit Test Project** /

xunit named `MappingObjects.Tests` to the `PopularPackages` solution.

2. In the `MappingObjects.Tests` project file, add project references to `MappingObjects.Models` and `MappingObjects.Mappers`, as shown in the following markup:

```
<ItemGroup> <ProjectReference Include= "..
\MappingObjects.Mappers
\MappingObjects.Mappers.csproj" />
<ProjectReference Include= "..
\MappingObjects.Models
\MappingObjects.Models.csproj" /> </
ItemGroup>
```

3. Build the `MappingObjects.Tests` project to restore packages and build referenced projects.
4. In the `MappingObjects.Tests` project, rename `UnitTest1.cs` to `AutoMapperTests.cs`.
5. Modify the contents of `AutoMapperTests.cs` to get the mapper and then assert that the mapping is complete, as shown in the following code:

```
using AutoMapper; // To use
MapperConfiguration. using
MappingObjects.Mappers; // To use
CartToSummaryMapper. namespace
MappingObjects.Tests; public class
AutoMapperTests { [Fact] public void
TestSummaryMapping() { MapperConfiguration
config =
CartToSummaryMapper.GetMapperConfiguration();
config.AssertConfigurationIsValid(); } }
```

6. Run the test:

- In Visual Studio, navigate to **Test | Run All Tests**.
- In VS Code, in **Terminal**, enter `dotnet test`.

9. Note the test fails because the `Total` member of the `Summary` view model is unmapped, as shown in *Figure 5.3*:

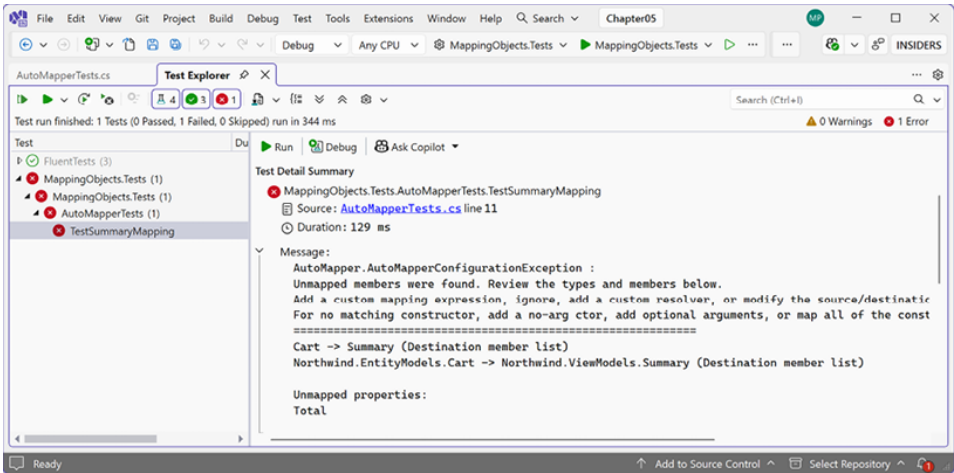


Figure 5.3: The test fails because the `Total` member is unmapped

1. In the `MappingObjects.Mappers` project, in the mapper configuration, after the mapping for the `FullName` member, add a mapping for the `Total` member, as shown highlighted in the following code:

```
cfg.CreateMap<Cart, Summary>() // Map the
    first and last names formatted to the full
    name. .ForMember(dest => dest.FullName,
    opt => opt.MapFrom(src =>
    string.Format("{0} {1}",
    src.Customer.FirstName,
    src.Customer.LastName) )) // We have
    removed a semi-colon from here. // Map the
    sum of items to the Total member.
```

```
.ForMember(dest => dest.Total, opt =>
opt.MapFrom( src => src.Items.Sum(item =>
item.UnitPrice * item.Quantity))); });
```

2. Run the test and note that, this time, it passes.

Performing live mappings between models

Now that we have validated the configuration of our mapping, we can use it in a live console app:

1. Use your preferred code editor to add a new **Console App** / `console` project named `MappingObjects.Console` to the `PopularPackages` solution.
2. In the `MappingObjects.Console` project, treat warnings as errors, globally and statically import the `System.Console` class, and add a project reference for the two class libraries, as shown in the following markup:

```
<ItemGroup> <ProjectReference Include= "..
\MappingObjects.Mappers
\MappingObjects.Mappers.csproj" />
<ProjectReference Include= "..
\MappingObjects.Models
\MappingObjects.Models.csproj" /> </
ItemGroup>
```

3. Build the `MappingObjects.Console` project.
4. In `Program.cs`, delete the existing statements, add statements to construct an example object model that represents a customer and their shopping cart with a couple of items, and then map it to a summary view

model to present to the user, as shown in the following code:

```
using AutoMapper; // To use
MapperConfiguration, IMapper. using
MappingObjects.Mappers; // To use
CartToSummaryMapper. using
Northwind.EntityModels; // To use
Customer, Cart, LineItem. using
Northwind.ViewModels; // To use Summary.
using System.Text; // To use Encoding. //
Set the console's output encoding to UTF-8
to support // Unicode characters like the
Euro currency symbol. OutputEncoding =
Encoding.UTF8; // Create an object model
from "entity" model types that // might
have come from a data store like SQL
Server. Cart cart = new( Customer: new(
FirstName: "John", LastName: "Smith" ),
Items: new() { new(ProductName: "Apples",
UnitPrice: 0.49M, Quantity: 10),
new(ProductName: "Bananas", UnitPrice:
0.99M, Quantity: 4) } ); WriteLine("***
Original data before mapping.");
WriteLine($"{cart.Customer}"); foreach
(LineItem item in cart.Items) {
WriteLine($" {item}"); } // Get the mapper
configuration for converting a Cart to a
Summary. MapperConfiguration config =
CartToSummaryMapper.GetMapperConfiguration();
// Create a mapper using the
configuration. IMapper mapper =
config.CreateMapper(); // Perform the
mapping. Summary summary =
mapper.Map<Cart, Summary>(cart); // Output
the result. WriteLine(); WriteLine("***
```



```
After mapping."); WriteLine($"Summary:
{summary.FullName} spent
{summary.Total:C}.");
```

5. Run the console app and note the successful result, as shown in the following code:

```
*** Original data before mapping. Customer
{ FirstName = John, LastName = Smith }
ListItem { ProductName = Apples, UnitPrice
= 0.49, Quantity = 10 } ListItem {
ProductName = Bananas, UnitPrice = 0.99,
Quantity = 4 } *** After mapping. Summary:
John Smith spent £8.86.
```

6. Optionally, write a unit test to perform a similar check as the preceding code to assert that the Summary has the correct full name and total.

Good practice: There is a debate about when AutoMapper should be used that you can read about in an article (which has more links at the bottom) at the following link: <https://www.anthonysteele.co.uk/AgainstAutoMapper.html>.

Learn more details at the following link: <https://automapper.org/>.

Mapping with extension methods and operators

As we saw in the rankings of package downloads, AutoMapper is very popular,

so it is good to learn how to use it so that you can work with it if necessary. But keep in mind that there are reasons to avoid mapping libraries like AutoMapper:

- AutoMapper hides the mapping logic, making debugging harder. If something goes wrong, you often have to dig into the AutoMapper configuration.
- AutoMapper relies on conventions and reflection, which means certain issues only appear at runtime.
- AutoMapper increases your project's dependency tree.

Instead of mapping with a third-party library, it is often better to define your own custom extension methods. Extension methods are direct, inline C# code. There's no reflection, no runtime mapping, and no unnecessary object creation. This makes them significantly faster for simple object-to-object transformations.

For example, a `ProductDto` class might be used to transmit a subset of properties from a `Product` entity between a web service and an app. You could create an extension method to simplify mapping, as shown in the following code:

```
using Northwind.EntityModels; // To use
Product. public static class ProductExtensions
{ public static ProductDto ToDto(this Product
product) { return new ProductDto { ProductId =
product.ProductId, ProductName =
product.ProductName, UnitPrice =
product.UnitPrice }; } }
```

Then you call the extension method on the entity instance, as shown in the following code:

```
ProductDto productDto = product.ToDto(); //
Clear and explicit.
```

For most everyday cases, extension methods are faster, clearer, and easier to maintain.

Alternatively, you could create an implicit casting operator, as shown in the following code:

```
public class ProductDto { public int ProductId
{ get; set; } public string ProductName { get;
set; } public decimal UnitPrice { get; set; }
// Implicit conversion operator from Product
to ProductDto. public static implicit operator
ProductDto(Product product) { if (product ==
null) return null; return new ProductDto {
ProductId = product.ProductId, ProductName =
product.ProductName, UnitPrice =
product.UnitPrice }; } }
```

Then you assign the entity instance to a DTO variable, as shown in the following code:

```
ProductDto productDto = product; // Implicit
casting.
```

But with implicit casting, it is less obvious what is happening, so I recommend the clearer calling of an extension method.

Making fluent assertions in unit testing

Assertions are at the heart of unit testing. They're the mechanism you use to verify that your code behaves as expected. Without assertions, tests are just bits of code that run. They wouldn't actually check anything or tell you when

something's wrong.

An assertion is a statement in a test that compares the actual outcome of some code to the expected outcome. If the comparison passes, the test continues or succeeds. If it fails, the test framework throws an exception and marks the test as failed.

FluentAssertions is a set of extension methods that make writing and reading the code in unit tests and the error messages of failing tests more similar to a natural human language like English.

Warning! Since version 8.0.0, released on January 14, 2025, there has been a license change, as described at the following links: <https://fluentassertions.com/releases/#800> and <https://github.com/fluentassertions/fluentassertions/pull/2943>. The owner dennisdooen commented: “v7 will remain free indefinitely and will still receive critical fixes. v8 will only require a license when you use it in non-commercial projects.” See the following links: <https://github.com/fluentassertions/fluentassertions/pull/2943#issuecomment-2590187813> and <https://github.com/fluentassertions/fluentassertions/pull/2943#issuecomment-2590286302>. Alternatively, there is a v7-forked repo named AwesomeAssertions: <https://www.nuget.org/packages/AwesomeAssertions/7.0.0>. You should be able to replace the package reference and

everything will continue to work.

It works with most unit testing frameworks, including xUnit. When you add a package reference for a test framework, FluentAssertions will automatically find the package and use it for throwing exceptions.

After importing the `FluentAssertions` namespace, call the `Should()` extension method on a variable and then one of the hundreds of other extension methods to make assertions in a human-readable way. You can chain multiple assertions using the `And()` extension method or have separate statements, each calling `Should()`.

Making assertions about strings

Let's start by making assertions about a single `string` value:

1. Use your preferred code editor to add a new **xUnit Test Project** / `xunit` named `FluentTests` to the `PopularPackages` solution.
2. In the `FluentTests` project, add a package reference to `FluentAssertions`, as highlighted in the following markup:

```
<ItemGroup> <PackageReference  
  Include="FluentAssertions" /> ...
```

3. Build the `FluentTests` project.
4. Rename `UnitTest1.cs` to `FluentAssertionsTests.cs`.
5. In `FluentAssertionsTests.cs`, import the namespace to make the `FluentAssertions` extension methods available and write a test method for a `string` value, as shown in the following code:

```
using FluentAssertions; // To use Should,  
And, HaveLength, and so on. namespace  
FluentTests; public class
```

```

FluentAssertionsTests { [Fact] public void
TestString() { string city = "London";
string expectedCity = "London";
city.Should().StartWith("Lo")
.And.EndWith("on") .And.Contain("do")
.And.HaveLength(6);
city.Should().NotNull()
.And.Be("London")
.And.BeSameAs(expectedCity)
.And.BeOfType<string>();
city.Length.Should().Be(6); } }

```

6. Run the test:

- In Visual Studio, navigate to **Test | Run All Tests**.
- In VS Code, in **Terminal**, enter `dotnet test`.

9. Note the test passes.

10. In the `TestString` method, for the `city` variable, delete the last `n` in `London`.

11. Run the test and note it fails, as shown in the following output:

```
Expected city "Londo" to end with "on".
```

12. Add the `n` back in `London`.

13. Run the test again to confirm the fix.

Making assertions about collections and arrays

Now let's continue by making assertions about collections and arrays:

1. In `FluentAssertionsTests.cs`, add a test method to explore collection assertions, as shown in the following code:

```
[Fact] public void TestCollections() {  
    string[] names = { "Alice", "Bob",  
        "Charlie" };  
    names.Should().HaveCountLessThan(4,  
        "because the maximum items should be 3 or  
        fewer"); names.Should().OnlyContain(name  
=> name.Length <= 6); }
```

2. Run the tests and note the collections test fails, as shown in the following output:

```
Expected names to contain only items  
matching (name.Length <= 6), but  
{"Charlie"} do(es) not match.
```

3. Change Charlie to Charly.
4. Run the tests and note that they succeed.

Making assertions about dates and times

Let's start by making assertions about date and time values:

1. In `FluentAssertionsTests.cs`, import the namespace for adding more extension methods for named months and other useful date/time-related functionality, as shown in the following code:

```
using FluentAssertions.Extensions; // To  
use February, March, and so on.
```

2. Add a test method to explore date/time assertions, as shown in the following code:

```
[Fact] public void TestDateTimes() {
    DateTime when = new( hour: 9, minute: 30,
        second: 0, day: 25, month: 3, year: 2024);
    when.Should().Be(25.March(2024).At(9,
        30));
    when.Should().BeOnOrAfter(23.March(2024));
    when.Should().NotBeSameDateAs(12.February(2024));
    when.Should().HaveYear(2024);
    DateTime due = new( hour: 11, minute: 0, second: 0,
        day: 25, month: 3, year: 2024);
    when.Should().BeAtLeast(2.Hours()).Before(due);
}
```

3. Run the tests and note the date/time test fails, as shown in the following output:

```
Expected when <2024-03-25 09:30:00> to be
at least 2h before <2024-03-25 11:00:00>,
but it is behind by 1h and 30m.
```

4. For the `due` variable, change the hour from `11` to `13`.
5. Run the tests and note that the date/time test succeeds.

Making assertions about equivalent objects

`BeEquivalentTo` is one of `FluentAssertions`' most powerful features. You use it when you want to check that two complex objects have equivalent data, without requiring them to be the same reference, the same runtime type, or even the same property order.

When you write:


```
actual.Should().BeEquivalentTo(expected);
```

FluentAssertions recursively compares all public properties and fields of `actual` and `expected` to ensure they contain the same values.

It's basically a deep structural comparison, not a shallow or reference equality check. For example:

```
User expected = new() { Id = 1, Name = "Alice" };
User actual = new() { Id = 1, Name = "Alice" };
actual.Should().BeEquivalentTo(expected); //
Passes.
```

Even though `actual` and `expected` are different objects in memory, this assertion passes because their data is equivalent. If you'd used `.Be(expected)` instead, it would have failed because `.Be()` checks for object identity or exact equality, using `Equals()` or reference equality.

Use `BeEquivalentTo` any time you want to verify that two objects represent the same data, regardless of whether they're the same instance. You can compare two different types as long as they have the same property names and values. For example, testing that your service correctly maps an entity to a DTO:

```
Product entity = new() { Id = 42, Name = "Tea", Price = 5.99m };
ProductDto dto = mapper.Map<ProductDto>(entity);
dto.Should().BeEquivalentTo(entity);
```

This is common when testing:

- DTO-to-Entity or Entity-to-DTO mappings

- API responses vs. expected models
- Query results vs. expected data structures

`BeEquivalentTo` is especially useful for collections. By default, it ignores ordering, so two lists with the same elements in different order will pass, as shown in the following code:

```
var expected = new[] { 1, 2, 3 }; var actual =  
new[] { 3, 1, 2 };  
actual.Should().BeEquivalentTo(expected); //  
Passes.
```

In the preceding case, equivalence means “same contents,” not “same order.”

If you do care about order, you can make that explicit, as shown in the following code:

```
actual.Should().BeEquivalentTo(expected,  
options => options.WithStrictOrdering());
```

`BeEquivalentTo` does not check:

- Reference equality (`object.ReferenceEquals`)
- Type equality (it can compare different types)
- Hidden or private members (only public)
- Behavior or methods, only data state

If you want to check that two variables reference the exact same instance, use `.BeSameAs(expected)` instead.

You can learn more details about `FluentAssertions` at the following link: <https://fluentassertions.com/>.

Making assertions in unit tests fluently makes your code more readable. In a similar way, defining validation rules fluently also improves your code for yourself and other developers.

Validating data

FluentValidation allows you to define strongly-typed validation rules in a human-readable way.

You create a validator for a type by inheriting from `AbstractValidator<T>`, where `T` is the type that you want to validate. In the constructor, you call the `RuleFor` method to define one or more rules. If a rule should run only in specified scenarios, then you call the `When` method.

Reviewing the built-in validators

FluentValidation ships with lots of useful built-in validator extension methods for defining rules, as shown in the following partial list, some of which you will explore in the coding task in this section:

- `Null`, `NotNull`, `Empty`, `NotEmpty`
- `Equal`, `NotEqual`
- `Length`, `MaxLength`, `MinLength`
- `LessThan`, `LessThanOrEqualTo`, `GreaterThan`, `GreaterThanOrEqualTo`
- `InclusiveBetween`, `ExclusiveBetween`
- `ScalePrecision`
- `Must` and `MustAsync` (aka predicate)
- `Matches` (aka regular expression), `EmailAddress`, `CreditCard`
- `IsInEnum`, `IsEnumName`

Customizing validation messages

There are a few extension methods that are used to customize the validation

messages' output when data fails to pass the rules:

- **WithName:** Change the name used for a property in the message.
- **WithSeverity:** Change the default severity from `Error` to `Warning` or some other level.
- **WithErrorCode:** Assign an error code that can be output in the message.
- **WithState:** Add some state that can be used in the message.
- **WithMessage:** Customize the format of the default message.

Defining a model and validator

Let's see an example of `FluentValidation` in action. You will create three projects:

- A class library for a model to validate that represents an order made by a customer.
- A class library for the validator for the model.
- A console app to perform a live validation.

Let's start:

1. Use your preferred code editor to add a new **Class Library** / `classlib` project named `FluentValidation.Models` to the `PopularPackages` solution.
2. In the `FluentValidation.Models` project, delete the file named `Class1.cs`.
3. In the `FluentValidation.Models` project, add a new class file named `CustomerLevel.cs` and modify its contents to define an enum with three customer levels, `Bronze`, `Silver`, and `Gold`, as shown in the following code:

```
namespace FluentValidation.Models; public
enum CustomerLevel { Bronze, Silver, Gold
```

```
}
```

4. In the `FluentValidation.Models` project, add a new class file named `Order.cs` and modify its contents, as shown in the following code:

```
namespace FluentValidation.Models; public
class Order { public long OrderId { get;
set; } public string? CustomerName { get;
set; } public string? CustomerEmail { get;
set; } public CustomerLevel CustomerLevel
{ get; set; } public decimal Total { get;
set; } public DateTime OrderDate { get;
set; } public DateTime ShipDate { get;
set; } }
```

5. Use your preferred code editor to add a new **Class Library** / classlib project named `FluentValidation.Validators` to the `PopularPackages` solution.
6. In the `FluentValidation.Validators` project, add a project reference to the `Models` project and a package reference to the `FluentValidation` package, as shown in the following markup:

```
<ItemGroup> <PackageReference
Include="FluentValidation" /> </ItemGroup>
<ItemGroup> <ProjectReference Include= "..
\FluentValidation.Models
\FluentValidation.Models.csproj" /> </
ItemGroup>
```

7. Build the `FluentValidation.Validators` project.
8. In the `FluentValidation.Validators` project, delete the file

named `Class1.cs`.

9. In the `FluentValidation.Validators` project, add a new class file named `OrderValidator.cs` and modify its contents, as shown in the following code:

```
using FluentValidation.Models; namespace
FluentValidation.Validators; public class
OrderValidator : AbstractValidator<Order>
{ public OrderValidator() { RuleFor(order
=> order.OrderId) .NotEmpty(); // Not
default(long) which is 0. RuleFor(order =>
order.CustomerName) .NotNull() .NotEmpty()
.WithName("Name"); // Use Name instead of
CustomerName in messages. RuleFor(order =>
order.CustomerName) .MinimumLength(5)
.WithSeverity(Severity.Warning);
RuleFor(order => order.CustomerEmail)
.NotEmpty() .EmailAddress(); RuleFor(order
=> order.CustomerLevel) .IsInEnum();
RuleFor(order => order.Total)
.GreaterThan(0); RuleFor(order =>
order.ShipDate) .GreaterThan(order =>
order.OrderDate); When(order =>
order.CustomerLevel == CustomerLevel.Gold,
() => { RuleFor(order =>
order.Total) .LessThan(50M); RuleFor(order
=> order.Total) .GreaterThanOrEqualTo(20M);
}).Otherwise(() => { RuleFor(order =>
order.Total) .LessThan(20M); }); } }
```

Testing the validator

Now we are ready to create a console app to test the validator on the model:

1. Use your preferred code editor to add a new **Console App** / console project named `FluentValidation.Console` to a `PopularPackages` solution.
2. In the `FluentValidation.Console` project, treat warnings as errors, globally and statically import the `System.Console` class, and add project references for `FluentValidation.Validators` and `FluentValidation.Models`, as shown in the following markup:

```
<ItemGroup> <ProjectReference Include= "..
\FluentValidation.Models
\FluentValidation.Models.csproj" />
<ProjectReference Include= "..
\FluentValidation.Validators
\FluentValidation.Validators.csproj" /> </
ItemGroup>
```

3. Build the `FluentValidation.Console` project to build referenced projects.
4. In `Program.cs`, delete the existing statements, and then add statements to create an order and validate it, as shown in the following code:

```
using FluentValidation.Models; // To use
Order. using FluentValidation.Results; //
To use ValidationResult. using
FluentValidation.Validators; // To use
OrderValidator. using
System.Globalization; // To use
CultureInfo. using System.Text; // To use
Encoding. OutputEncoding = Encoding.UTF8;
// Enable Euro symbol. // Control the
culture used for formatting of dates and
currency, // and for localizing error
```

```

messages to local language. Thread t =
Thread.CurrentThread; t.CurrentCulture =
CultureInfo.GetCultureInfo("en-US");
t.CurrentUICulture = t.CurrentCulture;
WriteLine($"Current culture:
{t.CurrentCulture.DisplayName}");
WriteLine(); Order order = new() { //
Start with a deliberately invalid order.
}; OrderValidator validator = new();
ValidationResult result =
validator.Validate(order); // Output the
order data. WriteLine($"CustomerName:
{order.CustomerName}");
WriteLine($"CustomerEmail:
{order.CustomerEmail}");
WriteLine($"CustomerLevel:
{order.CustomerLevel}");
WriteLine($"OrderId: {order.OrderId}");
WriteLine($"OrderDate:
{order.OrderDate}"); WriteLine($"ShipDate:
{order.ShipDate}"); WriteLine($"Total:
{order.Total:C}"); WriteLine(); // Output
if the order is valid and any rules that
were broken. WriteLine($"IsValid:
{result.IsValid}"); foreach (var item in
result.Errors) { WriteLine($"
{item.Severity}: {item.ErrorMessage}"); }

```

5. Run the console app and note the failed rules, as shown in the following output:

```

Current culture: English (United States)
CustomerName: CustomerEmail:
CustomerLevel: Bronze OrderId: 0

```



```
OrderDate: 01/01/0001 12:00:00 AM
ShipDate: 01/01/0001 12:00:00 AM Total:
$0.00 IsValid: False Error: 'Order Id'
must not be empty. Error: 'Name' must not
be empty. Error: 'Customer Email' must not
be empty. Error: 'Total' must be greater
than '0'. Error: 'Ship Date' must be
greater than '01/01/0001 12:00:00 AM'.
```

6. Comment out the two statements that set the culture to see the output in your local language and region. For example, if you are in France (fr-FR), it would look like the following:

```
Current culture: français (France)
CustomerName: CustomerEmail:
CustomerLevel: Bronze OrderId: 0
OrderDate: 01/01/0001 00:00:00 ShipDate:
01/01/0001 00:00:00 Total: 0,00 € IsValid:
False Error: 'Order Id' ne doit pas être
vide. Error: 'Name' ne doit pas avoir la
valeur null. Error: 'Customer Email' ne
doit pas être vide. Error: 'Total' doit
être plus grand que '0'. Error: 'Ship
Date' doit être plus grand que '01/01/0001
00:00:00'.
```

7. Set some property values for the order, as shown highlighted in the following code:

```
Order order = new() { OrderId = 10001,
CustomerName = "Abc", CustomerEmail =
"abc@example.com", CustomerLevel =
(CustomerLevel)4, OrderDate = new(2022,
```

```
month: 12, day: 1), ShipDate = new(2022,
month: 11, day: 5), Total = 49.99M };
```

8. Set the current culture to US English to make sure you see the same output as in this book. You can experiment with your own culture later.
9. Run the console app and note the failed rules, as shown in the following output:

```
Current culture: English (United States)
CustomerName: Abc CustomerEmail:
abc@example.com CustomerLevel: 4 OrderId:
10001 OrderDate: 12/1/2022 12:00:00 AM
ShipDate: 11/5/2022 12:00:00 AM Total:
$49.99 IsValid: False Warning: The length
of 'Customer Name' must be at least 5
characters. You entered 3 characters.
Error: 'Customer Email' is not a valid
email address. Error: 'Customer Level' has
a range of values which does not include
'4'. Error: 'Ship Date' must be greater
than '12/1/2022 12:00:00 AM'. Error:
'Total' must be less than '20'.
```

10. Modify some property values for the order, as shown highlighted in the following code:

```
Order order = new() { OrderId = 10001,
CustomerName = "Abcdef", CustomerEmail =
"abc@example.com", CustomerLevel =
CustomerLevel.Gold, OrderDate = new(2022,
month: 12, day: 1), ShipDate = new(2022,
month: 12, day: 5), // CustomerLevel is
Gold so Total can be >20. Total = 49.99M
```

```
};
```

11. Run the console app and note the order is now valid, as shown in the following output:

```
IsValid: True
```

Generating PDFs

One of the most common questions I get when teaching C# and .NET is, “What open-source library is available to generate PDF files?”

There are many licensed libraries for generating PDF files, but over the years, it has been difficult to find cross-platform open-source ones.

QuestPDF says, *“If you are consuming the QuestPDF library as a Direct Package Dependency for usage in a Closed Source software in the capacity of a for-profit company/individual with more than 1M USD annual gross revenue, you must purchase the QuestPDF Professional or Enterprise License, depending on the number of software developers. Please refer to the QuestPDF License and Pricing webpage for more details. (<https://www.questpdf.com/pricing.html>)”*

The older 2022.12.X release will always be available under the MIT license, free for commercial usage. If you want to support library development, please consider purchasing the Professional License for version 2023.1.X or later.

Using QuestPDF on Apple silicon Macs

QuestPDF uses SkiaSharp, which has implementations for Windows, Mac, and Linux operating systems. The console app that you create in this section to generate PDFs is therefore cross-platform. But on an Apple silicon Mac, like my Mac mini M1, I had to install the x64 version of .NET SDK and start the project

using `dotnet new -a x64`. This tells the .NET SDK to use the x64 architecture, otherwise the SkiaSharp libraries give an error because they have not yet been built to target ARM64.

Creating class libraries to generate PDF documents

Let's see an example of QuestPDF in action. You will create three projects:

- A class library for a model that represents a catalog of product categories with names and images.
- A class library for the document template.
- A console app to perform a live generation of a PDF file.

Let's start:

1. Use your preferred code editor to add a new **Class Library** / `classlib` project named `GeneratingPdf.Models` to the `PopularPackages` solution.
2. In the `GeneratingPdf.Models` project, delete the file named `Class1.cs`.
3. In the `GeneratingPdf.Models` project, add a new class file named `Category.cs` and modify its contents to define a class with two properties for the name and identifier of a category, as shown in the following code:

```
namespace GeneratingPdf.Models; public
class Category { public int CategoryId {
get; set; } public string CategoryName {
get; set; } = null!; }
```

Later, you will create an `images` folder with filenames that use the pattern

categoryN.jpeg, where N is a number from 1 to 8 that matches the CategoryId values.

1. In the GeneratingPdf.Models project, add a new class file named Catalog.cs and modify its contents to define a class with a property to store the eight categories, as shown in the following code:

```
namespace GeneratingPdf.Models; public
class Catalog { public List<Category>
Categories { get; set; } = null!; }
```

2. Use your preferred code editor to add a new **Class Library / classlib** project named GeneratingPdf.Document to the PopularPackages solution.
3. In the GeneratingPdf.Document project, add a package reference for QuestPDF and a project reference for the Models class library, as shown in the following markup:

```
<ItemGroup> <PackageReference
Include="QuestPDF" /> </ItemGroup>
<ItemGroup> <ProjectReference Include= "..
\GeneratingPdf.Models
\GeneratingPdf.Models.csproj" /> </
ItemGroup>
```

4. Build the GeneratingPdf.Document project.
5. In the GeneratingPdf.Document project, delete the file named Class1.cs.
6. In the GeneratingPdf.Document project, add a new class file named CatalogDocument.cs.

7. In `CatalogDocument.cs`, define a class that implements the `IDocument` interface to define a template with a header and a footer, and then output the eight categories, including name and image, as shown in the following code:

```
using GeneratingPdf.Models; // Catalog
using QuestPDF.Drawing; //
DocumentMetadata using QuestPDF.Fluent; //
Page using QuestPDF.Helpers; // Colors
using QuestPDF.Infrastructure; //
IDocument, IDocumentContainer namespace
GeneratingPdf.Document; public class
CatalogDocument : IDocument { public
Catalog Model { get; } public
CatalogDocument(Catalog model) { Model =
model; } public void
Compose(IDocumentContainer container) {
container .Page(page => { page.Margin(50 /
* points */); page.Header()
.Height(100).Background(Colors.Grey.Lighten1)
.AlignCenter().Text("Catalogue")
.Style(TextStyle.Default.FontSize(20));
page.Content()
.Background(Colors.Grey.Lighten3)
.Table(table => {
table.ColumnsDefinition(columns => {
columns.ConstantColumn(100);
columns.RelativeColumn(); }); foreach (var
item in Model.Categories) {
table.Cell().Text(item.CategoryName);
string imagePath = Path.Combine(
Environment.CurrentDirectory, "images",
$"category{item.CategoryId}.jpeg");
table.Cell().Image(imagePath); } });
page.Footer()
```

```
.Height(50).Background(Colors.Grey.Lighten1)
.AlignCenter().Text(x => {
x.CurrentPageNumber(); x.Span(" of ");
x.TotalPages(); }); }); } public
DocumentMetadata GetMetadata() =>
DocumentMetadata.Default; }
```

Creating a console app to generate PDF documents

Now we can create a console app project that will use the class libraries to generate a PDF document:

1. Use your preferred code editor to add a new **Console App** / console project named `GeneratingPdf.Console` to the `PopularPackages` solution.
2. In the `GeneratingPdf.Console` project, create an `images` folder and download the eight category images 1 to 8 from the following link to it: <https://github.com/markjprice/apps-services-net10/tree/master/images/Categories>.
3. If you are using Visual Studio or Rider, then the `images` folder and its files must be copied to the `GeneratingPdf.Console\bin\Debug\net10` folder:
4. In **Solution Explorer**, select all the images.
5. In **Properties**, set **Copy To Output Directory** to **Copy Always**.
6. Open the project file and note the `<ItemGroup>` entries that will copy the eight images to the correct folder, as partially shown in the following markup:

```
<ItemGroup> <None Update="images
\category1.jpeg">
<CopyToOutputDirectory>Always</
```

```
CopyToOutputDirectory> </None> ...
```

7. In the `GeneratingPdf.Console` project, treat warnings as errors, globally and statically import the `System.Console` class, and add a project reference for the `Document` template class library, as shown in the following markup:

```
<ItemGroup> <ProjectReference Include= "..  
  \GeneratingPdf.Document  
  \GeneratingPdf.Document.csproj" /> </  
ItemGroup>
```

8. Build the `GeneratingPdf.Console` project.
9. In `Program.cs`, delete the existing statements and then add statements to create a catalog model, pass it to a catalog document, generate a PDF file, and then attempt to open the file using the appropriate operating system command, as shown in the following code:

```
using GeneratingPdf.Document; // To use  
CatalogDocument. using  
GeneratingPdf.Models; // To use Catalog,  
Category. using QuestPDF.Fluent; // To use  
the GeneratePdf extension method. using  
QuestPDF.Infrastructure; // To use  
LicenseType. // For evaluation purposes,  
feel free to use the QuestPDF Community //  
License in a non-production environment.  
QuestPDF.Settings.License =  
LicenseType.Community; string filename =  
"catalog.pdf"; Catalog model = new() {  
Categories = new() { new() { CategoryId =  
1, CategoryName = "Beverages"}, new() {
```



```

CategoryId = 2, CategoryName =
"Condiments"}, new() { CategoryId = 3,
CategoryName = "Confections"}, new() {
CategoryId = 4, CategoryName = "Dairy
Products"}, new() { CategoryId = 5,
CategoryName = "Grains/Cereals"}, new() {
CategoryId = 6, CategoryName = "Meat/
Poultry"}, new() { CategoryId = 7,
CategoryName = "Produce"}, new() {
CategoryId = 8, CategoryName = "Seafood"
} }; CatalogDocument document =
new(model);
document.GeneratePdf(filename);
WriteLine("PDF catalog has been created:
{0}",
Path.Combine(Environment.CurrentDirectory,
filename)); try { if
(OperatingSystem.IsWindows()) {
System.Diagnostics.Process.Start("explorer.exe",
filename); } else { WriteLine("Open the
file manually."); } } catch (Exception ex)
{ WriteLine($" {ex.GetType()} says
{ex.Message}"); }

```

The `Process` class and its `Start` method should also be able to start processes on Mac and Linux, but getting the paths right can be tricky, so I've left that as an optional exercise for the reader. You can learn more about the `Process` class and its `Start` method at the following link: [`https://learn.microsoft.com/en-us/`](https://learn.microsoft.com/en-us/)

```
dotnet/api/  
system.diagnostics.process.start.
```

1. Run the console app and note the PDF file generated, as shown in *Figure 5.4*:

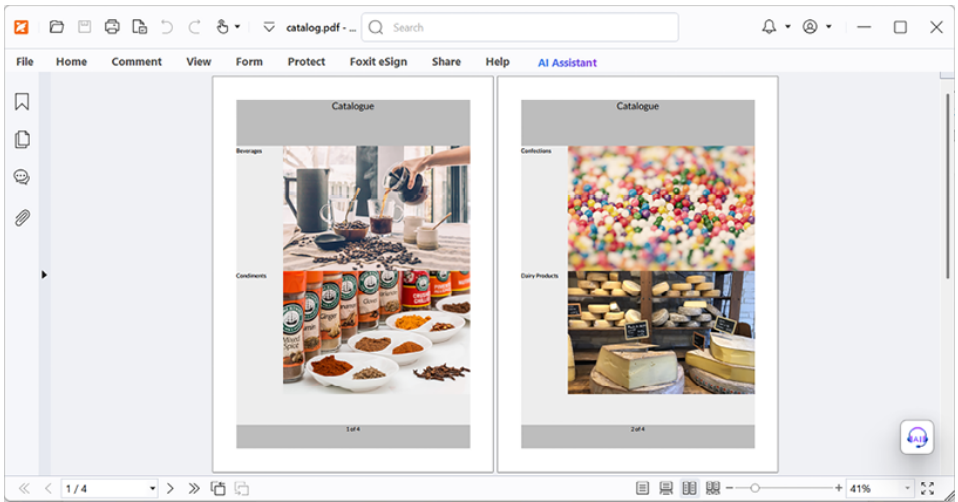


Figure 5.4: A PDF file generated from C# code

Learn more details at the following link:

<https://www.questpdf.com/>.

Practicing and exploring

Test your knowledge and understanding by answering some questions, getting some hands-on practice, and doing deeper research into the topics in this chapter.

Exercise 5.1 – Test your knowledge

Use the web to answer the following questions:

1. What is the most downloaded third-party NuGet package of all time?

2. What method do you call on the `ImageSharp Image` class to make a change like resizing the image or replacing colors with grayscale?
3. What is a key benefit of using `Serilog` for logging?
4. What is a `Serilog` sink?
5. Should you always use a package like `AutoMapper` to map between objects?
6. Which `FluentAssertions` method should you call to start a fluent assertion on a value?
7. Which `FluentAssertions` method should you call to assert that all items in a sequence conform to a condition, like a `string` item must have fewer than six characters?
8. Which `FluentValidation` class should you inherit from to define a custom validator?
9. With `FluentValidation`, how can you set a rule to only apply in certain conditions?
10. With `QuestPDF`, which interface must you implement to define a document for a PDF, and what methods of that interface must you implement?

Appendix A, Answers to the Test Your Knowledge Questions, is available to download from the following link: https://github.com/markjprice/apps-services-net10/blob/main/docs/B31467_Appendix%20A.pdf.

Exercise 5.2 – Explore topics

Use the links on the following page to learn more details about the topics covered in this chapter: <https://github.com/markjprice/apps-services-net10/blob/main/docs/book-links.md#chapter-5---implementing-popular-third->

party-libraries.

Summary

In this chapter, you explored some third-party libraries that are popular with .NET developers to perform functions, including:

- Manipulating images using a Microsoft-recommended third-party library named ImageSharp.
- Making text, numbers, dates, and times friendlier with Humanizer.
- Logging structured data with Serilog.
- Mapping between objects, for example, entity models to view models.
- Making fluent assertions in unit testing.
- Validating data in a local culture language-readable way.
- Generating a PDF file.

In the next chapter, we will learn how to handle internationalization with dates and times and localization, including a new type in .NET 8 or later for making it easier to unit test components with a dependency on the current time.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow this QR code:

https://packt.link/apps_and_services_dotnet10



Join .NETPro – It's Free

Staying sharp in .NET takes more than reading release notes. It requires real-world tips, proven patterns, and scalable solutions. That's what .NETPro, Packt's new newsletter, is all about.

Scan the QR code or visit the link to subscribe:

<https://landing.packtpub.com/dotnetpronewsletter/>



6

Handling Dates, Times, and Internationalization

This chapter is about some of the common types that are included with .NET. These include types to manipulate dates and times and implement internationalization, which includes globalization and localization.

When writing code to handle times, it is especially important to consider time zones. Bugs are often introduced because two times are compared in different time zones without taking that into account. It is important to understand the concept of **Coordinated Universal Time (UTC)** and to convert time values into UTC before performing time manipulation.

You should also be aware of any **Daylight Saving Time (DST)** adjustments that might be needed.

This chapter covers the following topics:

- Working with dates and times
- Working with time zones
- Working with cultures
- Working with Noda Time

Working with dates and times

After numbers and text, the next most popular types of data to work with are dates and times. The main types are as follows:

- `DateTime`: Represents a combined date and time value for a fixed point in time, optionally marked as `Local` (based on the system's time zone), `UTC` (Coordinated Universal Time), or `Unspecified` (no time zone context). However, it doesn't store the offset from UTC. This means if you pass a `DateTime` between systems in different time zones, you can lose context about what the original time actually meant. The problem with this type is that you can't tell which UTC offset was originally intended for a local time.
- `DateTimeOffset`: Represents a combined date and time value for a fixed point in time, including the offset from UTC (e.g., `+01:00`). This makes it unambiguous, and it always represents the same instant in time, regardless of where you are in the world.
- `TimeSpan`: Represents a duration of time. It's a measure of how much time passes between two instants, such as "2 hours, 15 minutes, 30 seconds", and it can be positive or negative. Its maximum range is about $\pm 10,675,199$ days (roughly $\pm 29,000$ years). Its resolution is 100 nanoseconds (a "tick"). It does not depend on time zones.

These types are often used together. For example, if you subtract one `DateTime` or `DateTimeOffset` value from another, the result is a `TimeSpan`. If you add a `TimeSpan` to a `DateTime` or `DateTimeOffset`, then the result is a `DateTime` or `DateTimeOffset` value.

Good practice: `DateTime` exists mainly for historical reasons and is fine for local-only apps or when you know exactly how the time is used. `DateTimeOffset` was introduced to solve real-world problems with time zone ambiguity,

serialization, and cross-system consistency. Use `DateTimeOffset` for most modern applications, especially anything involving data exchange or time zones. Use `DateTime` only if you specifically need compatibility with legacy code or APIs that expect it.

Specifying date and time values

A common way to create a date and time value is to specify individual values for the date and time components, like day and hour, as described in *Table 6.1*:

Date/time parameter		
year	1999	
month		
day	the number of days in that month	
hour	23	
minute	59	
second	59	
microsecond		
nanosecond		
nanosecond		

Table 6.1: Parameters for specifying date and time values

For example, to instantiate a `DateTime` that represents when .NET 11 might be released for **General Availability**, as shown in the following code:

```
DateTime dotnet11GA = new(year: 2026, month: 11, day: 10, hour: 11, minute: 0, second: 0);
```

Good practice: The preceding code example might make you think, “What time zone does the value represent?” This is the big problem with

`DateTime` and why it is good practice to avoid it in favor of `DateTimeOffset`, which stores the time zone too. We will look at this issue in more detail later in this chapter.

An alternative is to provide the value as a `string` to be parsed, but this can be misinterpreted depending on the default culture of the thread. For example, in the UK, dates are specified as day/month/year, whereas in the US, dates are specified as month/day/year.

Let's see what you might want to do with dates and times:

1. Use your preferred code editor to create a new project, as defined in the following list:
 - Project template: **Console App** / `console`
 - Project file and folder: `WorkingWithTime`
 - Solution file and folder: `Internationalization`
 - **Do not use top-level statements**: Cleared
 - **Enable native AOT publish**: Cleared
7. In the project file, treat warnings as errors, and add an element to statically and globally import the `System.Console` class.
8. Add a new class file named `Program.Helpers.cs` and replace its contents, as shown in the following code:

```
using System.Globalization; // To use
CultureInfo. partial class Program {
private static void
ConfigureConsole(string culture = "en-US",
bool overrideComputerCulture = true) { //
To enable special characters like Euro
currency symbol. OutputEncoding =
System.Text.Encoding.UTF8; Thread t =
Thread.CurrentThread; if
(overrideComputerCulture) {
```

```

t.CurrentCulture =
CultureInfo.GetCultureInfo(culture);
t.CurrentUICulture = t.CurrentCulture; }
CultureInfo ci = t.CurrentCulture;
WriteLine($"Current culture:
{ci.DisplayName}"); WriteLine($"Short date
pattern: {
ci.DateTimeFormat.ShortDatePattern}");
WriteLine($"Long date pattern: {
ci.DateTimeFormat.LongDatePattern}");
WriteLine(); } private static void
SectionTitle(string title) { ConsoleColor
previousColor = ForegroundColor;
ForegroundColor = ConsoleColor.DarkYellow;
WriteLine($"*** {title}"); ForegroundColor
= previousColor; } }

```

9. In `Program.cs`, delete the existing statements, and then add statements to initialize some special date/time values, as shown in the following code:

```

ConfigureConsole(); // Defaults to en-US
culture. SectionTitle("Specifying date and
time values");
WriteLine($"DateTime.MinValue:
{DateTime.MinValue}");
WriteLine($"DateTime.MaxValue:
{DateTime.MaxValue}");
WriteLine($"DateTime.UnixEpoch:
{DateTime.UnixEpoch}");
WriteLine($"DateTime.Now:
{DateTime.Now}");
WriteLine($"DateTime.Today:
{DateTime.Today}");

```

```
WriteLine($"DateTime.Today:
{DateTime.Today:d}");
WriteLine($"DateTime.Today:
{DateTime.Today:D}");
```

10. Run the code, and note the results, as shown in the following output:

```
Current culture: English (United States)
Short date pattern: M/d/yyyy Long date
pattern: dddd, MMMM d, yyyy *** Specifying
date and time values DateTime.MinValue:
1/1/0001 12:00:00 AM DateTime.MaxValue:
12/31/9999 11:59:59 PM DateTime.UnixEpoch:
1/1/1970 12:00:00 AM DateTime.Now:
5/30/2023 9:18:05 AM DateTime.Today:
5/30/2023 12:00:00 AM DateTime.Today:
5/30/2023 DateTime.Today: Tuesday, May 30,
2023
```

The date and time formats output are determined by the culture settings of your console app. We called the `ConfigureConsole` method to make sure we all see the same default output in US English.

1. In `Program.cs`, at the top of the statement that calls `ConfigureConsole`, set the parameter to not override your local computer's culture, as shown in the following code:

```
ConfigureConsole(overrideComputerCulture:
false);
```

2. Run the code, and note the output is localized to your computer's culture.
3. In `Program.cs`, set the parameter to specify alternative languages, like French in Canada (`fr-CA`) or English in Great Britain (`en-GB`), as shown in the following code:

```
ConfigureConsole("fr-CA");
```

There is a table of common culture codes at the following link: https://en.wikipedia.org/wiki/Language_localisation#Language_tag

1. Run the code, and note that the output is localized to the specified culture.
2. Reset the console configuration back to the default so that it uses US English culture, as shown in the following code:

```
ConfigureConsole(); // Defaults to en-US culture.
```

Formatting date and time values

You have just seen that dates and times have default formats based on the current culture.

You can take complete control of date and time formatting using custom format codes, as shown in *Table 6.2*:

Description		
Date part separator. Varies by culture; for example, en-US uses /, but fr-FR uses - (dash)		
Escape character. Useful if you want to use a special format code as a		

	literal character; for example, <code>h \h m \m</code> would format a time of 9:30 am as <code>9 h 30 m</code>		
	Time part separator. Varies by culture; for example, <code>en-US</code> uses <code>:</code> , but <code>fr-FR</code> uses <code>.</code> (dot)		
	The day of the month, from <code>1</code> to <code>31</code> , or with a leading zero from <code>01</code> through <code>31</code>		
	The abbreviated or full name of the day of the week, for example, <code>Mon</code> or <code>Monday</code> , localized for the current culture		
	The tenths of a second, hundredths of a second, or milliseconds		
	The period or era, for example, <code>A D</code>		
	The hour, using a 12-hour clock from <code>1</code> to <code>12</code> , or from <code>01</code> to <code>12</code>		
	The hour, using a 24-hour clock from <code>0</code> to <code>23</code> , or from <code>01</code> to <code>23</code>		
	Time zone information. <code>null</code> for an unspecified time zone, <code>Z</code> for UTC, and a value like <code>-8:00</code> for local time adjusted from UTC		
	The minute, from <code>0</code> through <code>59</code> , or with a leading zero from <code>00</code> through <code>59</code>		
	The month, from <code>1</code> through <code>12</code> , or with a leading zero from <code>01</code> through <code>12</code>		
	The abbreviated or full name of the month, for example, <code>Jan</code> or <code>January</code> , localized for the current culture		
	The second, from <code>0</code> through <code>59</code> , or with a leading zero from <code>00</code> through <code>59</code>		
	The first or both characters of the AM/PM designator		
	The year of the current century, from <code>0</code> through <code>99</code> , or with a leading zero from <code>00</code> through <code>99</code>		
	The year with a minimum of three digits, and as many as needed. For example, 1 A.D. is <code>001</code> . The first sacking of Rome was in <code>410</code> . The year this book was published is <code>2023</code>		
	The year as a four- or five-digit number		
	Hours offset from UTC, with no leading zeros, or with leading zeros		
	Hours and minutes offset from UTC, with a leading zero, for example, <code>+04:30</code> .		

Table 6.2: Custom format codes for date and time values

A full list of custom format code can be found at the following link: <https://learn.microsoft.com/en-us/dotnet/standard/base-types/custom-date-and-time-format-strings>.

You can apply standard date and time formatting using simpler format codes, like the `d` and `D` we used in the code example, as shown in Table 6.3:

Descripti	code		
Short date pattern. Varies by culture; for example, en-US uses M/d/yyyy and fr-FR uses dd/MM/yyyy			
Long date pattern. Varies by culture; for example, en-US uses mmmm, MMMM d, yyyy and fr-FR uses mmmm, dd MMMM yyyy			
Full date/time pattern (short time – hours and minutes). Varies by culture			
Full date/time pattern (long time – hours, minutes, seconds, and AM/PM). Varies by culture			
A standardized pattern, defined by ISO 8601, suitable to serialize date/time values for roundtrips even between different development platforms and languages, for example, 2023-05-30T13:45:30.0000000-08:00			
RFC1123 pattern			
Short time pattern. Varies by culture; for example, en-US uses h:mm ff and fr-FR uses HH:mm			
Long time pattern. Varies by culture; for example, en-US uses h:mm:ss ff and fr-FR uses HH:mm:ss			
Universal sortable date/time pattern, for example, 2009-06-15 13:45:30Z			
Universal full date/time pattern. Varies by culture; for example, en-US might be Monday, June 15, 2009 8:45:30 PM.			

Table 6.3: Standard format codes for date and time values

A full list of format code can be found at the following link: <https://learn.microsoft.com/en-us/dotnet/standard/base-types/standard-date-and-time-format-strings>.

Let's run some examples of formatting date and time values:

1. In `Program.cs`, add statements to define Christmas Day in 2024 and display it in various ways, as shown in the following code:

```
DateTime xmas = new(year: 2024, month: 12,
day: 25); WriteLine($"Christmas (default
format): {xmas}"); WriteLine($"Christmas
(custom short format): {xmas:ddd d/M/
yy}"); WriteLine($"Christmas (custom long
format): { xmas:dddd, dd MMMM yyyy}");
WriteLine($"Christmas (standard long
format): {xmas:D}"); WriteLine($"Christmas
(sortable): {xmas:u}");
WriteLine($"Christmas is in month
{xmas.Month} of the year.");
WriteLine($"Christmas is day
{xmas.DayOfYear} of {xmas.Year}.");
WriteLine($"Christmas {xmas.Year} is on a
{xmas.DayOfWeek}.");
```

2. Run the code, and note the results, as shown in the following output:

```
Christmas (default format): 12/25/2024
12:00:00 AM Christmas (custom short
format): Wed, 25/12/24 Christmas (custom
```

```
long format): Wednesday, 25 December 2024
Christmas (standard long format):
Wednesday, December 25, 2024 Christmas
(sortable): 2024-12-25 00:00:00Z Christmas
is in month 12 of the year. Christmas is
day 360 of 2024. Christmas 2024 is on a
Wednesday.
```

3. Disable overriding your computer culture or pass a specific culture code, like French in France, as shown in the following code:

```
ConfigureConsole("fr-FR"); // Defaults to
en-US culture.
```

4. Run the code, and note that the results should be localized to that culture.
5. Reset the console configuration back to the default of US English.

Date and time calculations

Now, let's try performing simple calculations with date and time values:

1. In `Program.cs`, add statements to perform addition and subtraction with Christmas 2024, as shown in the following code:

```
SectionTitle("Date and time
calculations"); DateTime beforeXmas =
xmas.Subtract(TimeSpan.FromDays(12));
DateTime afterXmas = xmas.AddDays(12);
WriteLine($"12 days before Christmas:
{beforeXmas:d}"); WriteLine($"12 days
after Christmas: {afterXmas:d}"); TimeSpan
untilXmas = xmas - DateTime.Now;
WriteLine($"Now: {DateTime.Now}");
```



```
WriteLine($"There are {untilXmas.Days}  
days and {untilXmas.Hours } hours until  
Christmas {xmas.Year."}); WriteLine("There  
are {untilXmas.TotalHours:N0} hours " +  
$"until Christmas {xmas.Year}.");
```

2. Run the code, and note the results, as shown in the following output:

```
*** Date and time calculations 12 days  
before Christmas: 12/13/2024 12 days after  
Christmas: 1/6/2025 Now: 5/30/2023 1:57:01  
PM There are 574 days and 10 hours until  
Christmas 2024. There are 13,786 hours  
until Christmas 2024.
```

3. Add statements to define the time on Christmas Day that your children (or dog or cat or iguana?) might wake up to open presents, and display it in various ways, as shown in the following code:

```
DateTime kidsWakeUp = new( year: 2024,  
month: 12, day: 25, hour: 6, minute: 30,  
second: 0); WriteLine($"Kids wake up:  
{kidsWakeUp}"); WriteLine($"The kids woke  
me up at {  
kidsWakeUp.ToShortTimeString()}");
```

4. Run the code, and note the results, as shown in the following output:

```
Kids wake up: 25/12/2024 06:30:00 AM The  
kids woke me up at 06:30 AM
```

Microseconds and nanoseconds

In earlier versions of .NET, the smallest unit of time measurement was a tick. A tick is 100 nanoseconds, so developers used to have to do the calculation for nanoseconds themselves. .NET 7 introduced millisecond and microsecond parameters to constructors, and microsecond and nanosecond properties to the `DateTime`, `DateTimeOffset`, `TimeSpan`, and `TimeOnly` types.

Let's see some example code:

1. In `Program.cs`, add statements to construct a date and time value with more precision than was previously possible and to display its value, as shown in the following code:

```
SectionTitle("Milli-, micro-, and nanoseconds"); DateTime preciseTime = new(
    year: 2022, month: 11, day: 8, hour: 12,
    minute: 0, second: 0, millisecond: 6,
    microsecond: 999);
WriteLine($"Millisecond:
{preciseTime.Millisecond}, Microsecond: {
preciseTime.Microsecond}, Nanosecond:
{preciseTime.Nanosecond}"); preciseTime =
DateTime.UtcNow; // Nanosecond value will
be 0 to 900 in 100 nanosecond increments.
WriteLine($"Millisecond:
{preciseTime.Millisecond}, Microsecond: {
preciseTime.Microsecond}, Nanosecond:
{preciseTime.Nanosecond}");
```

2. Run the code, and note the results, as shown in the following output:

```
*** Milli-, micro-, and nanoseconds
Millisecond: 6, Microsecond: 999,
Nanosecond: 0 Millisecond: 243,
```

Microsecond: 958, Nanosecond: 400

Warning! You can't specify nanoseconds in the constructor or set the `Nanosecond` property, because this is a read-only property, but you can call the `AddTicks` method, for example, `AddTicks(1)` to add the equivalent of 100 nanoseconds for each tick.

Globalization with dates and times

The current culture controls how dates and times are formatted and parsed. Let's write some code to see how to control the current culture to control how dates and times are formatted and parsed:

1. At the top of `Program.cs`, import the namespace to work with globalization, as shown in the following code:

```
using System.Globalization; // To use
CultureInfo.
```

2. Add statements to show the current culture that is used to display date and time values, and then parse the United States' Independence Day and display it in various ways, as shown in the following code:

```
SectionTitle("Globalization with dates and
times"); // Same as
Thread.CurrentThread.CurrentCulture.
WriteLine($"Current culture:
{CultureInfo.CurrentCulture.Name}");
string textDate = "4 July 2024"; DateTime
independenceDay =
```

```

DateTime.Parse(textDate);
WriteLine($"Text: {textDate}, DateTime:
{independenceDay:d MMMM}"); textDate =
"7/4/2024"; independenceDay =
DateTime.Parse(textDate);
WriteLine($"Text: {textDate}, DateTime:
{independenceDay:d MMMM}"); // Explicitly
override the current culture by setting a
provider. independenceDay =
DateTime.Parse(textDate, provider:
CultureInfo.GetCultureInfo("en-US"));
WriteLine($"Text: {textDate}, DateTime:
{independenceDay:d MMMM}");

```

Good practice: Although you can create a `CultureInfo` instance using its constructor, unless you need to make changes to it, you should get a read-only shared instance by calling the `GetCultureInfo` method.

1. At the top of `Program.cs`, set the culture to British English, as shown in the following code:

```

ConfigureConsole("en-GB");

```

2. Run the code, and note the results, as shown in the following output:

```

*** Globalization with dates and times
Current culture is: en-GB Text: 4 July
2024, DateTime: 4 July Text: 7/4/2024,
DateTime: 7 April Text: 7/4/2024,

```

DateTime: 4 July

When the current culture is *English (Great Britain)*, if a date is given as 4 July 2024, then it is correctly parsed regardless of whether the current culture is British or American. But if the date is given as 7/4/2024, then it is parsed as 7 April. You can override the current culture by specifying the correct culture as a provider when parsing, as shown in the third example above.

1. Add statements to loop from the year 2023 to 2028, displaying if the year is a leap year and how many days there are in February, and then showing if Christmas and Independence Day are during DST, as shown in the following code:

```
for (int year = 2023; year <= 2028; year++) { Write($"{year} is a leap year: {DateTime.IsLeapYear(year)}. "); WriteLine($"There are {DateTime.DaysInMonth(year: year, month: 2)} days in February {year}."); } WriteLine($"Is Christmas daylight saving time? { xmas.IsDaylightSavingTime()}"); WriteLine($"Is July 4th daylight saving time? { independenceDay.IsDaylightSavingTime()}");
```

2. Run the code, and note the results, as shown in the following output:

```
2023 is a leap year: False. There are 28
days in February 2023. 2024 is a leap
year: True. There are 29 days in February
2024. 2025 is a leap year: False. There
are 28 days in February 2025. 2026 is a
leap year: False. There are 28 days in
February 2026. 2027 is a leap year: False.
There are 28 days in February 2027. 2028
is a leap year: True. There are 29 days in
February 2028. Is Christmas daylight
saving time? False Is July 4th daylight
saving time? True
```

Complexities of Daylight Saving Time (DST)

Each country has its own rules for what day and what hour DST happens. These rules are encoded by .NET so that it can adjust automatically when needed. Of course, if the leader of a country issues an Executive Order suddenly declaring the end of DST, software will need to be upgraded to encode that new information. While .NET has robust, built-in mechanisms to handle time zone conversions and DST adjustments automatically, these features don't eliminate the complexities and potential pitfalls that DST creates. Relying on them blindly can lead to subtle but significant bugs.

When you get the current local time, .NET consults the OS to see if DST is in effect for the machine's configured time zone and returns the correctly adjusted time. For example, if you ask for the time in Europe/London, .NET knows to use British Summer Time (BST) in the summer and Greenwich Mean Time (GMT) in the winter.

Automatic adjustments can't resolve logical issues, especially when dealing with stored times, future schedules, or times from different zones.

In the US in springtime, the clocks “spring” forward one hour at 2 AM. In the fall, they “fall” back one hour at 2 AM. Wikipedia explains this at the following link: https://en.wikipedia.org/wiki/Daylight_saving_time_in_the_United_States.

In the UK in springtime, the clocks spring forward one hour at 1 AM. In the autumn, they fall back one hour at 2 AM. The UK government explains this at the following link: <https://www.gov.uk/when-do-the-clocks-change>.

Imagine that you need to set an alarm to wake you up at 1:30 AM to catch a flight from Heathrow airport in the UK. Your flight happens to depart on the day that DST takes effect.

In the UK spring, the clocks are at 12:59 AM, and then the next minute they spring forward to 2:00 AM. 1:30 AM never happens, your alarm does not go off, and you miss your flight! 1:30 AM is an invalid time in .NET, and if you try to store that value in a variable, it will throw an exception.

In the UK autumn, the clocks are at 1:59 AM, and then the next minute, they fall back to 1:00 AM and repeat that hour. In this case, 1:30 AM happens twice.

If you have a timestamp like “2024-10-27 01:30:00”, you don’t know if it refers to the first 1:30 AM (in BST) or the second 1:30 AM (in GMT). .NET’s `TimeZoneInfo` class has methods like `IsAmbiguousTime()` to detect this, but it’s up to the developer to decide how to handle it. Should you assume the first occurrence, the second, or ask the user for clarification?

DST is not used in all countries; it is also determined by hemisphere, and politics plays a role. For example, the United States is currently debating whether it should make DST permanent. They might decide to leave the decision up to individual states. It could all get extra confusing for Americans over the next few years.

Localizing the DayOfWeek enum

DayOfWeek is an `enum`, so it cannot be localized as you might expect or hope. Its `string` values are hardcoded in English, as shown in the following code:

```
namespace System { public enum DayOfWeek {  
    Sunday = 0, Monday = 1, Tuesday = 2, Wednesday  
    = 3, Thursday = 4, Friday = 5, Saturday = 6 }  
}
```

There are two solutions to this problem.

First, you could apply the `dddd` date format code to a whole date value. For example,

```
WriteLine($"The day of the week is  
{DateTime.Now:dddd}.");
```

This is a simple solution, though rather hacky.

Second, the official solution is to use the `GetDayName` helper method of the `DateTimeFormatInfo` class to convert a `DayOfWeek` value into a localized `string` for output as text.

Let's see an example of the problem and both solutions:

1. In `Program.cs`, add statements to explicitly set the current culture to Danish, and then output the current day of the week in that culture, as shown in the following code:

```
SectionTitle("Localizing the DayOfWeek  
enum"); CultureInfo previousCulture =  
Thread.CurrentThread.CurrentCulture; //  
Explicitly set culture to Danish
```



```

(Denmark) .
Thread.CurrentThread.CurrentCulture =
CultureInfo.GetCultureInfo("da-DK"); //
DayOfWeek is not localized to Danish.
WriteLine("Culture:
{Thread.CurrentThread.CurrentCulture
.NativeName}, DayOfWeek:
{DateTime.Now.DayOfWeek}"); // Use dddd
format code to get day of the week
localized. WriteLine($"Culture:
{Thread.CurrentThread.CurrentCulture
.NativeName}, DayOfWeek:
{DateTime.Now:dddd}"); // Use GetDayName
method to get day of the week localized.
WriteLine("Culture:
{Thread.CurrentThread.CurrentCulture
.NativeName}, DayOfWeek:
{DateTimeFormatInfo.CurrentInfo
.GetDayName(DateTime.Now.DayOfWeek)}");
Thread.CurrentThread.CurrentCulture =
previousCulture;

```

2. Run the code, and note the results, as shown in the following output:

```

*** Localizing the DayOfWeek enum Culture:
dansk (Danmark), DayOfWeek: Thursday
Culture: dansk (Danmark), DayOfWeek:
torsdag Culture: dansk (Danmark),
DayOfWeek: torsdag

```

Working with only a date or a time

.NET 6 introduced some new types when working with only a date value or only

a time value, named `DateOnly` and `TimeOnly`.

These are better than using a `DateTime` value with zeros for the time values to store a date-only value, because they are type-safe and avoid misuse.

`DateOnly` also maps better to database column types, for example, a `date` column in SQL Server. `TimeOnly` is good for setting alarms and scheduling regular meetings or the opening hours for an organization, and it maps to a `time` column in SQL Server.

Let's use them to plan a release party for .NET 11, probably on Tuesday, November 10, 2026:

1. In `Program.cs`, add statements to define the .NET 11 release party and a time for it to start, and then combine the two values to make a calendar entry so that we don't miss it, as shown in the following code:

```
SectionTitle("Working with only a date or  
a time"); DateOnly party = new(year: 2026,  
month: 11, day: 10); WriteLine($"The .NET  
11 release party is on  
{party.ToLongDateString()}."); TimeOnly  
starts = new(hour: 11, minute: 30);  
WriteLine($"The party starts at  
{starts}."); DateTime calendarEntry =  
party.ToDateTime(starts); WriteLine($"Add  
to your calendar: {calendarEntry}.");
```

2. Run the code and note the results, as shown in the following output:

```
*** Working with only a date or a time The  
.NET 11 release party is on Tuesday,  
November 10, 2026. The party starts at  
11:30 AM. Add to your calendar: 11/10/2026  
11:30:00 AM.
```

Warning! `TimeOnly` is a poor choice for an event because it does not include information about the time zone. For events, we need to understand and handle time zones, and you will learn about these in a later section.

Getting date/time formatting information

Globalization is the overall concept. This section is about where the globalization rules are defined. Each culture has its own date/time formatting rules. These are defined in the `DateTimeFormat` property of a `CultureInfo` instance.

Let's output some commonly used information:

1. In `Program.cs`, add statements to get the date/time formatting information for the current culture and output some of its most useful properties, as shown in the following code:

```
SectionTitle("Working with date/time  
formats"); DateTimeFormatInfo dtfi =  
DateTimeFormatInfo.CurrentInfo; // Or use  
Thread.CurrentThread.CurrentCulture.DateTimeFormat  
WriteLine($"Date separator:  
{dtfi.DateSeparator}"); WriteLine($"Time  
separator: {dtfi.TimeSeparator}");  
WriteLine($"Long date pattern:  
{dtfi.LongDatePattern}");  
WriteLine($"Short date pattern:  
{dtfi.ShortDatePattern}");  
WriteLine($"Long time pattern:  
{dtfi.LongTimePattern}");  
WriteLine($"Short time pattern:  
{dtfi.ShortTimePattern}"); Write("Day
```

```
names:"); for (int i = 0; i <
dtfi.DayNames.Length - 1; i++) { Write($"
{dtfi.GetDayName((DayOfWeek)i)}"); }
WriteLine(); Write("Month names:"); for
(int i = 1; i < dtfi.MonthNames.Length; i+
+) { Write($" {dtfi.GetMonthName(i)}"); }
WriteLine();
```

2. Run the code, and note the results, as shown in the following output:

```
*** Working with date/time formats Date
separator: / Time separator: : Long date
pattern: dddd, MMMM d, yyyy Short date
pattern: M/d/yyyy Long time pattern:
h:mm:ss tt Short time pattern: h:mm tt Day
names: Sunday Monday Tuesday Wednesday
Thursday Friday Month names: January
February March April May June July August
September October November December
```

3. Change the culture to something else, run the code, and note the results.

Unit testing with a time provider

Writing unit tests for components that need the current time is tricky because the time is constantly changing!

Imagine you want visitors to your e-commerce website to get a 20% discount if they make an order at the weekend. During workdays, they pay full price. How can we test this functionality?

To control the time used in unit tests, .NET 8 introduced the `TimeProvider` class.

Let's start defining a function to perform this calculation:

1. In the Internationalization solution, add a new **Class Library / classlib** project named `TimeFunctionsLib`.
2. In the `TimeFunctionsLib` project, rename `Class1.cs` to `DiscountService.cs`.
3. In `DiscountService.cs`, define a function to perform the calculation, as shown in the following code:

```
namespace Northwind.Services; public class
DiscountService { public decimal
GetDiscount() { // This has a dependency
on the current time provided by the
system. var now = DateTime.UtcNow; return
now.DayOfWeek switch { DayOfWeek.Saturday
or DayOfWeek.Sunday => 0.2M, _ => 0M }; }
}
```

4. In the Internationalization solution, add a new **xUnit Test Project**/xunit project named `TestingWithTimeProvider`.
5. In the `TestingWithTimeProvider` project, add a reference to the `TimeFunctionsLib` project, as shown in the following markup:

```
<ItemGroup> <ProjectReference Include= "..
\TimeFunctionsLib\TimeFunctionsLib.csproj"
/> </ItemGroup>
```

6. Build the `TestingWithTimeProvider` project.
7. In the `TestingWithTimeProvider` project, rename `Test1.cs` to `TimeTests.cs`.
8. In `TimeTests.cs`, modify the statements to import the namespace for the discount service, and then define two tests, one for workdays and one for weekends, as shown in the following code:

```

using Northwind.Services; // To use
DiscountService. namespace
TestingWithTimeProvider; public class
TimeTests { [Fact] public void
TestDiscountDuringWorkdays() { // Arrange
DiscountService service = new(); // Act
decimal discount = service.GetDiscount();
// Assert Assert.Equal(0M, discount); }
[Fact] public void
TestDiscountDuringWeekends() {
DiscountService service = new(); decimal
discount = service.GetDiscount();
Assert.Equal(0.2M, discount); } }

```

9. Run the two tests, and note that only one can ever succeed at any one time. If you run the tests during workdays, the weekend test will fail. If you run the tests during the weekend, the workday test will fail!

Now that you've seen the problem, how can we solve it?

Implementing a time provider

The way Microsoft solves it is by having each team that creates .NET libraries define its own internal `ISystemClock` interface with, at a minimum, a property named `UtcNow`, and sometimes other members, along with implementations that typically use the built-in system clock but are all slightly different. A typical example is shown in the following code:

```

using System; namespace
Microsoft.Extensions.Internal { public
interface ISystemClock { DateTimeOffset UtcNow
{ get; } } public class SystemClock :
ISystemClock { public DateTimeOffset UtcNow {
return DateTimeOffset.UtcNow; } } }

```

Finally, with .NET 8, the core .NET team has introduced a proper equivalent of the preceding code with an implementation that uses the system clock.

Unfortunately, they do not define an interface. Instead, they define an abstract class named `TimeProvider`.

Let's use it:

1. In the `TimeFunctionsLib` project, in `DiscountService.cs`, comment out the use of the `UtcNow` property, and add a statement to add a constructor-injected service, as highlighted in the following code:

```
namespace Northwind.Services; public class
DiscountService { private TimeProvider
_timeProvider; public
DiscountService(TimeProvider timeProvider)
{ _timeProvider = timeProvider; } public
decimal GetDiscount() { // This has a
dependency on the current time provided by
the system. // var now = DateTime.UtcNow;
var now = _timeProvider.GetUtcNow(); //
This has a dependency on the current time
provided by the system. return
now.DayOfWeek switch { DayOfWeek.Saturday
or DayOfWeek.Sunday => 0.2M, _ => 0M }; }
}
```

2. In the `TestingWithTimeProvider` project, in `TimeTests.cs`, add statements to both tests to show how we could use the new `TimeProvider` and its `System` property (which would still have a dependency on the system clock!), as shown in the following code:

```
// This would use the .NET 8 or later
dependency service, // but its
implementation is still the system clock.
```

```
DiscountService service =  
new (TimeProvider.System);
```

3. In the `TestingWithTimeProvider` project, add a reference to `Moq`, a package to mock dependencies, as shown in the following markup:

```
<!-- The newest version before the  
controversy. --> <PackageReference  
Include="Moq" />
```

4. In `TimeTests.cs`, import the namespace to use the `Mock.Of<T>` extension method, as shown in the following code:

```
using Moq; // To use Mock.Of<T> method.
```

5. In the `TestDiscountDuringWorkdays` method, comment out the statement that uses the `System` provider, and replace it with statements to mock a time provider that always returns a fixed date and time during workdays, as highlighted in the following code:

```
TimeProvider timeProvider =  
Mock.Of<TimeProvider>(); // Mock the time  
provider so it always returns the date of  
// 2023-11-07 09:30:00 UTC which is a  
Tuesday. Mock.Get(timeProvider).Setup(s =>  
s.GetUtcNow()).Returns( new  
DateTimeOffset(year: 2023, month: 11, day:  
7, hour: 9, minute: 30, second: 0, offset:  
TimeSpan.Zero)); DiscountService service =  
new (timeProvider);
```


6. In the `TestDiscountDuringWeekends` method, comment out the statement that used the `System` provider and replace it with statements to mock a time provider that always returns a fixed date and time at the weekend, as highlighted in the following code:

```
TimeProvider timeProvider =  
Mock.Of<TimeProvider>(); // Mock the time  
provider so it always returns the date of  
// 2023-11-04 09:30:00 UTC which is a  
Saturday. Mock.Get(timeProvider).Setup(s  
=> s.GetUtcNow()).Returns( new  
DateTimeOffset(year: 2023, month: 11, day:  
4, hour: 9, minute: 30, second: 0, offset:  
TimeSpan.Zero)); DiscountService service =  
new(timeProvider);
```

7. Run the unit tests, and note that they both succeed.

In the preceding sections I have mentioned the importance of time zones. Let's now see how to work with them in .NET.

Working with time zones

In the code example about the .NET release party in the *Working with only a date or a time* section, using a `TimeOnly` was not actually a good idea because the `TimeOnly` value did not include information about the time zone. It is only useful if you are in the correct time zone. `TimeOnly` is, therefore, a poor choice for an event. For events, we need to understand and handle time zones.

Understanding DateTime and TimeZoneInfo

The `DateTime` class has many useful members related to time zones, as shown in *Table 6.4*:

Description		
<code>Now</code> property		Value that represents the current date and time in the local time zone
<code>UtcNow</code> property		Value that represents the current date and time in the UTC time zone
<code>Kind</code> property		Kind value that indicates whether the <code>DateTime</code> value is Unspecified, UTC, or Local
<code>IsDaylightSavingTime</code> method		Indicates if the <code>DateTime</code> value is during DST
<code>ConvertToUtc</code> method		Converts the <code>DateTime</code> value to the equivalent local time
<code>ConvertToLocalTime</code> method		Converts the local <code>DateTime</code> value to the equivalent UTC time.

Table 6.4: `DateTime` members related to time zones

The `TimeZoneInfo` class has many useful members, as shown in *Table 6.5*:

Description		
<code>Id</code> property		Uniquely identifies the time zone
<code>LocalTime</code> property		<code>TimeZoneInfo</code> value that represents the current local time zone. Varies depending on where the code executes
<code>UtcTime</code> property		<code>TimeZoneInfo</code> value that represents the UTC time zone
<code>StandardName</code> property		Name of the time zone when Daylight Saving is not active
<code>DaylightName</code> property		Name of the time zone when Daylight Saving is active
<code>DisplayName</code> property		Display name of the time zone
<code>BaseUtcOffset</code> property		Represents the difference between this time zone and the UTC time zone, ignoring any potential Daylight Saving adjustments
<code>SupportsDaylightSaving</code> property		Indicates whether this time zone has Daylight Saving adjustments
<code>ConvertToTime</code> method		Converts a <code>DateTime</code> value to another <code>DateTime</code> value in a different time zone. You can specify the source and destination time zones

<code>ConvertUtcDateTimeToDateTimeOffset</code>	Converts a <code>DateTime</code> value in the UTC time zone to a <code>DateTimeOffset</code> value in a specified time zone
<code>ConvertDateTimeOffsetToUtcDateTime</code>	Converts a <code>DateTimeOffset</code> value in a specified time zone to a <code>DateTime</code> value in the UTC time zone
<code>IsDaylightSaving</code>	Returns a <code>bool</code> indicating whether the <code>DateTime</code> value is in Daylight Saving
<code>GetSystemTimeZoneInfo</code>	Returns a collection of time zones registered with the operating system.

Table 6.5: `TimeZoneInfo` useful members

Some database providers for EF Core only allow you to store `DateTime` values that use the `Kind` property to determine whether it is UTC, so you might need to convert to and from `DateTimeOffset` if you need to work with these values.

Exploring `DateTime` and `TimeZoneInfo`

Use the `TimeZoneInfo` class to work with time zones:

1. Use your preferred code editor to add a new **Console App** / `console` project named `WorkingWithTimeZones` to the `Internationalization` solution.
2. In Visual Studio, set **Startup Project** to **Current selection**.
3. Treat warnings as errors, and statically and globally import the `System.Console` class.
4. Add a new class file named `Program.Helpers.cs`.
5. Modify its contents to define some helper methods to output a section title in a visually different way, output a list of all time zones in the current system, and output details about a `DateTime` or `TimeZoneInfo` object, as shown in the following code:

```

using System.Collections.ObjectModel; //
To use ReadOnlyCollection<T> partial class
Program { private static void
SectionTitle(string title) { ConsoleColor
previousColor = ForegroundColor;
ForegroundColor = ConsoleColor.DarkYellow;
WriteLine($"*** {title}"); ForegroundColor
= previousColor; } private static void
OutputTimeZones() { // get the time zones
registered with the OS
ReadOnlyCollection<TimeZoneInfo> zones =
TimeZoneInfo.GetSystemTimeZones();
WriteLine($"*** {zones.Count} time
zones:"); // order the time zones by Id
instead of DisplayName foreach
(TimeZoneInfo zone in zones.OrderBy(z =>
z.Id)) { WriteLine($" {zone.Id}"); } }
private static void
OutputDateTime(DateTime dateTime, string
title) { SectionTitle(title);
WriteLine($"Value: {dateTime}");
WriteLine($"Kind: {dateTime.Kind}");
WriteLine($"IsDaylightSavingTime:
{dateTime.IsDaylightSavingTime()}");
WriteLine($"ToLocalTime():
{dateTime.ToLocalTime()}");
WriteLine($"ToUniversalTime():
{dateTime.ToUniversalTime()}"); } private
static void OutputTimeZone(TimeZoneInfo
zone, string title) { SectionTitle(title);
WriteLine($"Id: {zone.Id}");
WriteLine($"IsDaylightSavingTime(DateTime.Now):
{
zone.IsDaylightSavingTime(DateTime.Now)}");
WriteLine($"StandardName:

```

```

{zone.StandardName}");
WriteLine($"DaylightName:
{zone.DaylightName}");
WriteLine($"BaseUtcOffset:
{zone.BaseUtcOffset}"); } private static
string GetCurrentZoneName(TimeZoneInfo
zone, DateTime when) { // time zone names
change if Daylight Saving time is active
// e.g. GMT Standard Time becomes GMT
Summer Time return
zone.IsDaylightSavingTime(when) ?
zone.DaylightName : zone.StandardName; } }

```

6. In `Program.cs`, delete the existing statements. Add statements to output the current date and time in the local and UTC time zones, and then output details about the local and UTC time zones, as shown in the following code:

```

OutputTimeZones();
OutputDateTime(DateTime.Now,
"DateTime.Now");
OutputDateTime(DateTime.UtcNow,
"DateTime.UtcNow");
OutputTimeZone(TimeZoneInfo.Local,
"TimeZoneInfo.Local");
OutputTimeZone(TimeZoneInfo.Utc,
"TimeZoneInfo.Utc");

```

7. Run the console app and note the results, including the time zones registered on your operating system (there are 141 on my Windows 11 laptop), and that it is currently 4:17 PM on 31 May 2022 in England, meaning I am in the GMT Standard Time zone. However, because DST is active, it is currently known as GMT Summer Time, which is one hour

ahead of UTC, as shown in the following partial output:

```
*** 141 time zones: Afghanistan Standard
Time Alaskan Standard Time ... West
Pacific Standard Time Yakutsk Standard
Time Yukon Standard Time *** DateTime.Now
Value: 31/05/2022 16:17:03 Kind: Local
IsDaylightSavingTime: True ToLocalTime():
31/05/2022 16:17:03 ToUniversalTime():
31/05/2022 15:17:03 *** DateTime.UtcNow
Value: 31/05/2022 15:17:03 Kind: Utc
IsDaylightSavingTime: False ToLocalTime():
31/05/2022 16:17:03 ToUniversalTime():
31/05/2022 15:17:03 *** TimeZoneInfo.Local
Id: GMT Standard Time
IsDaylightSavingTime(DateTime.Now): True
StandardName: GMT Standard Time
DaylightName: GMT Summer Time
BaseUtcOffset: 00:00:00 ***
TimeZoneInfo.Utc Id: UTC
IsDaylightSavingTime(DateTime.Now): False
StandardName: Coordinated Universal Time
DaylightName: Coordinated Universal Time
BaseUtcOffset: 00:00:00
```

The `BaseUtcOffset` of the **GMT Standard Time** zone is zero because, normally, Daylight Saving is not active. That is why it is prefixed with `Base`.

1. In `Program.cs`, add statements to prompt the user to enter a time zone (using Eastern Standard Time as a default), get that time zone, output details about it, and then compare a time entered by the user with the

equivalent time in the other time zone, and catch potential exceptions, as shown in the following code:

```
Write("Enter a time zone or press Enter  
for US East Coast: "); string zoneId =  
ReadLine()!; if  
(string.IsNullOrEmpty(zoneId)) { zoneId =  
"Eastern Standard Time"; } try {  
    TimeZoneInfo otherZone =  
    TimeZoneInfo.FindSystemTimeZoneById(zoneId);  
    OutputTimeZone(otherZone,  
    $"TimeZoneInfo.FindSystemTimeZoneById(\"{zoneId}\")\n",  
    SectionTitle($"What's the time in  
{zoneId}?"); Write("Enter a local time or  
press Enter for now: "); string? timeText  
    = ReadLine(); DateTime localTime; if  
(string.IsNullOrEmpty(timeText) || !  
    DateTime.TryParse(timeText, out  
    localTime)) { localTime = DateTime.Now; }  
    DateTime otherZoneTime =  
    TimeZoneInfo.ConvertTime( dateTime:  
    localTime, sourceTimeZone:  
    TimeZoneInfo.Local, destinationTimeZone:  
    otherZone); WriteLine($"{localTime}  
{GetCurrentZoneName(TimeZoneInfo.Local,  
    localTime)} is {otherZoneTime}  
{GetCurrentZoneName(otherZone,  
    otherZoneTime)}."); } catch  
(TimeZoneNotFoundException) {  
    WriteLine($"The {zoneId} zone cannot be  
    found on the local system."); } catch  
(InvalidTimeZoneException) {  
    WriteLine($"The {zoneId} zone contains  
    invalid or missing data."); } catch  
(System.Security.SecurityException) {
```

```
WriteLine("The application does not have  
permission to read time zone  
information."); } catch  
(OutOfMemoryException) { WriteLine($"Not  
enough memory is available to load  
information on the {zoneId} zone."); }
```

2. Run the console app, press *Enter* for US East Coast, then enter 12:30pm for the local time, and note the results, as shown in the following output:

```
Enter a time zone or press Enter for US  
East Coast: ***  
TimeZoneInfo.FindSystemTimeZoneById("Eastern  
Standard Time") Id: Eastern Standard Time  
IsDaylightSavingTime(DateTime.Now): True  
StandardName: Eastern Standard Time  
DaylightName: Eastern Summer Time  
BaseUtcOffset: -05:00:00 *** What's the  
time in Eastern Standard Time? Enter a  
local time or press Enter for now: 12:30pm  
31/05/2023 12:30:00 GMT Summer Time is  
31/05/2023 07:30:00 Eastern Summer Time.
```

My local time zone is GMT Standard Time, so there is currently a five-hour time difference between me and the US East Coast. Your local time zone will be different.

1. Run the console app, copy one of the time zones to the clipboard, paste it at the prompt, and then press *Enter* for the local time. Note the results, as

shown in the following output:

```
Enter a time zone or press Enter for US
East Coast: AUS Eastern Standard Time ***
TimeZoneInfo.FindSystemTimeZoneById("AUS
Eastern Standard Time") Id: AUS Eastern
Standard Time
IsDaylightSavingTime(DateTime.Now): False
StandardName: AUS Eastern Standard Time
DaylightName: AUS Eastern Summer Time
BaseUtcOffset: 10:00:00 *** What's the
time in AUS Eastern Standard Time? Enter a
local time or press Enter for now:
31/05/2023 17:00:04 GMT Summer Time is
01/06/2023 02:00:04 AUS Eastern Standard
Time.
```

Sydney, Australia, is currently nine hours ahead, so at about 5 PM for me, it is about 2 AM on the following day for them.

That's a lot to learn about dates, times, and time zones. But we aren't done yet. Now, we need to look at the wider topic of cultures, which are a combination of language and region and do not just affect date and time formatting.

Working with cultures

Internationalization is the process of enabling your code to correctly run all over the world. It has two parts, **globalization** and **localization**, and both of them are about working with cultures.

Globalization is about writing your code to accommodate multiple languages and region combinations. The combination of a language and a region is known

as a culture. It is important for your code to know both the language and region because, for example, the date and currency formats are different in Quebec and Paris, despite them both using the French language.

There are **International Organization for Standardization (ISO)** codes for all culture combinations. For example, in the code `da-DK`, `da` indicates the Danish language and `DK` indicates the Denmark region, and in the code `fr-CA`, `fr` indicates the French language and `CA` indicates the Canada region.

ISO is not just an acronym. ISO is a reference to the Greek word *isos* (which means *equal*). You can see a list of ISO culture codes at the following link:

<https://loneolfonline.net/list-net-culture-country-codes/>.

Localization is about customizing the user interface to support a language, for example, changing the label of a button to **Close** (`en`) or **Fermer** (`fr`). Since localization is more about the language, it doesn't always need to know about the region, although, ironically enough, the words *standardization* (`en-US`) and *standardisation* (`en-GB`) suggest otherwise.

Good practice: I am not a professional translator of software user interfaces, so take all examples in this chapter as general guidance. My research into French user interface labeling common practice led me to the following links, but it would be best to hire a professional if you are not a native language speaker: <https://french.stackexchange.com/questions/12969/translation-of-it-terms-like-close-next-search-etc> and <https://www.linguee.com/english-french/translation/close>

```
+button.html.
```

Detecting and changing the current culture

Internationalization is a huge topic on which thousand-page books have been written. In this section, you will get a brief introduction to the basics, using the `CultureInfo` and `RegionInfo` types in the `System.Globalization` namespace.

Let's write some code:

1. Use your preferred code editor to add a new **Console App** / `console` project named `WorkingWithCultures` to the `Internationalization` solution.
2. In the project file, treat warnings as errors, and then statically and globally import the `System.Console` class, and globally import the `System.Globalization` namespace so that we can use the `CultureInfo` class, as shown in the following markup:

```
<ItemGroup> <Using  
  Include="System.Console" Static="true" />  
<Using Include="System.Globalization" />  
</ItemGroup>
```

3. Add a new class file named `Program.Helpers.cs`, and modify its contents to add a method to the partial `Program` class that will output information about the cultures used for globalization and localization, as shown in the following code:

```
partial class Program { private static  
  void OutputCultures(string title) {  
    ConsoleColor previousColor =
```

```
ForegroundColor; ForegroundColor =
ConsoleColor.DarkYellow; WriteLine($"***
{title}"); // Get the cultures from the
current thread. CultureInfo globalization
= CultureInfo.CurrentCulture; CultureInfo
localization =
CultureInfo.CurrentUICulture;
WriteLine($"The current globalization
culture is { globalization.Name}:
{globalization.DisplayName}");
WriteLine($"The current localization
culture is { localization.Name}:
{localization.DisplayName}");
WriteLine($"Days of the week:
{string.Join(", ",
globalization.DateTimeFormat.DayNames)}");
WriteLine($"Months of the year:
{string.Join(", ",
globalization.DateTimeFormat.MonthNames //
Some have 13 months; most 12, and the last
is empty. .TakeWhile(month => !
string.IsNullOrEmpty(month))}");
WriteLine($"1st day of this year: {new
DateTime( year: DateTime.Today.Year,
month: 1, day: 1) .ToString("D",
globalization)}"); WriteLine($"Number
group separator: {globalization
.NumberFormat.NumberGroupSeparator}");
WriteLine($"Number decimal separator:
{globalization
.NumberFormat.NumberDecimalSeparator}");
RegionInfo region =
new(globalization.LCID);
WriteLine($"Currency symbol:
{region.CurrencySymbol}");
```

```

WriteLine($"Currency name:
{region.CurrencyNativeName} ({
region.CurrencyEnglishName})");
WriteLine($"IsMetric: {region.IsMetric}");
WriteLine(); ForegroundColor =
previousColor; } }

```

4. In `Program.cs`, delete the existing statements, and add statements to set the output encoding of the console to support Unicode. Then, output information about the globalization and localization cultures. Finally, prompt the user to enter a new culture code and show how that affects the formatting of common values, such as dates and currency, as shown in the following code:

```

// To enable special characters like €.
OutputEncoding =
System.Text.Encoding.UTF8;
OutputCultures("Current culture");
WriteLine("Example ISO culture codes:");
string[] cultureCodes = { "da-DK", "en-
GB", "en-US", "fa-IR", "fr-CA", "fr-FR",
"he-IL", "pl-PL", "sl-SI" }; foreach
(string code in cultureCodes) {
CultureInfo culture =
CultureInfo.GetCultureInfo(code);
WriteLine($" {culture.Name}:
{culture.EnglishName} / {
culture.NativeName}"); } WriteLine();
Write("Enter an ISO culture code: ");
string? cultureCode = ReadLine(); if
(string.IsNullOrEmpty(cultureCode)) {
cultureCode = "en-US"; } CultureInfo ci;
try { ci =
CultureInfo.GetCultureInfo(cultureCode); }

```

```

catch (CultureNotFoundException) {
WriteLine($"Culture code not found:
{cultureCode}"); WriteLine("Exiting the
app."); return; } // change the current
cultures on the thread
CultureInfo.CurrentCulture = ci;
CultureInfo.CurrentUICulture = ci;
OutputCultures("After changing the current
culture"); Write("Enter your name: ");
string? name = ReadLine(); if
(string.IsNullOrEmpty(name)) { name =
"Bob"; } Write("Enter your date of birth:
"); string? dobText = ReadLine(); if
(string.IsNullOrEmpty(dobText)) { //
If they do not enter a DOB then use //
sensible defaults for their culture.
dobText = ci.Name switch { "en-US" or "fr-
CA" => "1/27/1990", "da-DK" or "fr-FR" or
"pl-PL" => "27/1/1990", "fa-IR" =>
"1990/1/27", _ => "1/27/1990" }; }
Write("Enter your salary: "); string?
salaryText = ReadLine(); if
(string.IsNullOrEmpty(salaryText)) {
salaryText = "34500"; } DateTime dob =
DateTime.Parse(dobText); int minutes =
(int)DateTime.Today.Subtract(dob).TotalMinutes;
decimal salary =
decimal.Parse(salaryText);
WriteLine($"{name} was born on a
{dob:dddd}. {name} is { minutes:N0}
minutes old. {name} earns {salary:C}.");

```

When you run an application, it automatically sets its thread to use the culture of the operating system. I am running my code in London, UK,

so the thread is set to English (Great Britain).

The code prompts the user to enter an alternative ISO code. This allows your application to replace the default culture at runtime.

The application then uses standard format codes to output the day of the week using format code `dddd`, the number of minutes with thousand separators using format code `N0`, and the salary with the currency symbol. These adapt automatically, based on the thread's culture.

1. Run the code and enter `en-US` for the ISO code (or press *Enter*), and then enter some sample data, including a date in a format valid for US English, as shown in the following output:

```
*** Current culture The current
globalization culture is en-GB: English
(United Kingdom) The current localization
culture is en-GB: English (United Kingdom)
Days of the week: Sunday, Monday, Tuesday,
Wednesday, Thursday, Friday, Saturday
Months of the year: January, February,
March, April, May, June, July, August,
September, October, November, December 1st
day of this year: 01 January 2023 Number
group separator: , Number decimal
separator: . Currency symbol: £ Currency
name: British Pound (British Pound)
IsMetric: True Example ISO culture codes:
da-DK: Danish (Denmark) / dansk (Danmark)
en-GB: English (United Kingdom) / English
(United Kingdom) en-US: English (United
States) / English (United States) fa-IR:
Persian (Iran) / ایران (ایران) fr-CA: French
(Canada) / français (Canada) fr-FR: French
(France) / français (France) he-IL: Hebrew
```

```
(Israel) / לארשי (עברית) pl-PL: Polish
(Poland) / polski (Polska) sl-SI:
Slovenian (Slovenia) / slovenščina
(Slovenija) Enter an ISO culture code: en-
US *** After changing the current culture
The current globalization culture is en-
US: English (United States) The current
localization culture is en-US: English
(United States) Days of the week: Sunday,
Monday, Tuesday, Wednesday, Thursday,
Friday, Saturday Months of the year:
January, February, March, April, May,
June, July, August, September, October,
November, December 1st day of this year:
Sunday, January 1, 2023 Number group
separator: , Number decimal separator: .
Currency symbol: $ Currency name: US
Dollar (US Dollar) IsMetric: False Enter
your name: Alice Enter your date of birth:
3/30/1967 Enter your salary: 34500 Alice
was born on a Thursday. Alice is
29,541,600 minutes old. Alice earns
$34,500.00
```

2. Run the code again, and try Danish in Denmark (da-DK), as shown in the following output:

```
Enter an ISO culture code: da-DK *** After
changing the current culture The current
globalization culture is da-DK: dansk
(Danmark) The current localization culture
is da-DK: dansk (Danmark) Days of the
week: søndag, mandag, tirsdag, onsdag,
torsdag, fredag, lørdag Months of the
```



```
year: januar, februar, marts, april, maj,
juni, juli, august, september, oktober,
november, december 1st day of this year:
søndag den 1. januar 2023 Number group
separator: . Number decimal separator: ,
Currency symbol: kr. Currency name: dansk
krone (Danish Krone) IsMetric: True Enter
your name: Mikkel Enter your date of
birth: 16/3/1980 Enter your salary: 65000
Mikkel was born on a søndag. Mikkel is
22.723.200 minutes old. Mikkel earns
65.000,00 kr..
```

In this example, only the date and salary are globalized into Danish. The rest of the text is hardcoded as English. Later, we will translate that English text into other languages. For now, let's see some other differences between cultures.

1. Run the code again, and try Polish in Poland (pl-PL). Note that the grammar rules in Polish make the day number possessive for the month name, so the month `styczeń` becomes `stycznia`, as shown in the following output:

```
The current globalization culture is pl-
PL: polski (Polska) ... Months of the
year: styczeń, luty, marzec, kwiecień,
maj, czerwiec, lipiec, sierpień, wrzesień,
październik, listopad, grudzień 1st day of
this year: niedziela, 1 stycznia 2023 ...
Enter your name: Bob Enter your date of
birth: 1972/4/16 Enter your salary: 50000
Bob was born on a niedziela. Bob is 26 886
240 minutes old. Bob earns 50 000,00 zł.
```

2. Run the code again, and try Persian in Iran (fa-IR). Note that dates in Iran must be specified as year/month/day, and that this year (2023) is the year 1401 in the Persian calendar, as shown in the following output:

```
The current globalization culture is fa-IR: (فارسی ایران) The current localization culture is fa-IR: (فارسی ایران) Days of the week: یکشنبه, دوشنبه, سه‌شنبه, چهارشنبه, پنجشنبه, شنبه, جمعه Months of the year: فروردین, اردیبهشت, خرداد, تیر, مرداد, شهریور, مهر, آبان, آذر, دی 1401 day of this year: اسفند 1 شنبه, 11 Number group separator: , Number decimal separator: , Currency symbol: ریال Currency name: (Iranian Rial) ریال ایران IsMetric: True Enter your name: Cyrus Enter your date of birth: 1372/4/16 Enter your salary: 50000 Cyrus was born on a چهارشنبه. Cyrus is 15,723,360 minutes old. Cyrus earns 50,000ریال.
```

Although I tried to confirm with a Persian reader whether this example is correct, due to factors like right-to-left languages being tricky to work with in console apps and copying and pasting from a console window into a word processor, I apologize in advance to my Persian readers if this example is all messed up!

Temporarily using the invariant culture

Sometimes, you might need to temporarily use a different culture without

switching the current thread to that culture. For example, when automatically generating documents, queries, and commands that include data values, you might need to ignore your current culture and use a more standardized culture. For this purpose, you can use the invariant culture, which is based on US English.

For example, you might need to generate a JSON document with a decimal number value and format the number with two decimal places, as shown in the following code:

```
decimal price = 54321.99M; string document = $
$"" { "price": "{{price:N2}}" } "";
```

If you were to execute this on a Slovenian computer, you would get the following output:

```
{ "price": "54.321,99" }
```

If you then tried to insert this JSON document into a cloud database, it would fail because it would not understand the number format that uses commas for decimals and dots for groups.

So you can override the current culture and specify the invariant culture only when outputting the number as a `string` value, as shown in the following code:

```
decimal price = 54321.99M; string document = $
$"" { "price": "{{price.ToString("N2",
CultureInfo.InvariantCulture)}}" } "";
```

If you were to execute this on a Slovenian (or any other culture) computer, you would now get the following output, which would be successfully recognized by

a cloud database and would not throw exceptions:

```
{ "price": "54,321.99" }
```

Now, let's see how to localize text from one language to another so that the label prompts are in the correct language for the current culture.

Localizing your user interface

A localized application is divided into two parts:

- An assembly containing code that is the same for all locales and contains resources for when no other resource file is found.
- One or more assemblies that contain the user interface resources, which are different for different locales. These are known as **satellite assemblies**.

This model allows the initial application to be deployed with default invariant resources, and over time, additional satellite assemblies can be deployed as the resources are translated. In the coding task for this section, you will create a console app with embedded invariant culture, and satellite assemblies for Danish, French, French-Canadian, Polish, and Iranian (Persian). To add more cultures in the future, just follow the same steps.

User interface resources include any text for messages, logs, dialog boxes, buttons, labels, or even filenames of images, videos, and so on. Resource files are XML files with the `.resx` extension. The filename includes a culture code, for example, `PacktResources.en-GB.resx` or `PacktResources.da-DK.resx`.

If a resource file or individual entry is missing, the automatic culture fallback search path for resources goes from specific culture (language and region) to neutral culture (language only) to invariant culture (supposedly independent but, basically, US English). If the current thread culture is `en-AU` (Australian

English), then it will search for the resource file in the following order:

1. Australian English: `PacktResources.en-AU.resx`
2. Neutral English: `PacktResources.en.resx`
3. Invariant: `PacktResources.resx`

Defining and loading resources

To load resources from these satellite assemblies, we use some standard .NET types named `IStringLocalizer<T>` and `IStringLocalizerFactory`. Implementations of these are loaded from the .NET generic host as dependency services:

1. In the `WorkingWithCultures` project, add package references to Microsoft extensions to work with generic hosting and localization, as shown in the following markup:

```
<ItemGroup> <PackageReference  
  Include="Microsoft.Extensions.Hosting" />  
<PackageReference  
  Include="Microsoft.Extensions.Localization"  
/> </ItemGroup>
```

2. Build the `WorkingWithCultures` project to restore packages.
3. In the project folder, create a new folder named `Resources`.
4. In the `Resources` folder, add a new XML file named `PacktResources.resx`, and modify the contents to contain default invariant language resources (usually equivalent to US English), as shown in the following markup:

```
<?xml version="1.0" encoding="utf-8"?>  
<root> <data name="EnterYourDob"  
  xml:space="preserve"> <value>Enter your
```

```

date of birth: </value> </data> <data
name="EnterYourName" xml:space="preserve">
<value>Enter your name: </value> </data>
<data name="EnterYourSalary"
xml:space="preserve"> <value>Enter your
salary: </value> </data> <data
name="PersonDetails" xml:space="preserve">
<value>{0} was born on a {1:dddd}. {0} is
{2:N0} minutes old. {0} earns {3:C}.</
value> </data> </root>

```

5. In the `WorkingWithCultures` project folder, add a new class file named `PacktResources.cs` that will load text resources for the user interface, as shown in the following code:

```

using Microsoft.Extensions.Localization;
// To use IStringLocalizer and so on.
public class PacktResources { private
readonly IStringLocalizer<PacktResources>
localizer = null!; public
PacktResources(IStringLocalizer<PacktResources>
localizer) { this.localizer = localizer; }
public string? GetEnterYourNamePrompt() {
string resourceStringName =
"EnterYourName"; // 1. Get the
LocalizedString object. LocalizedString
localizedString =
localizer[resourceStringName]; // 2. Check
if the resource string was found. if
(localizedString.ResourceNotFound) {
ConsoleColor previousColor =
ForegroundColor; ForegroundColor =
ConsoleColor.Red; WriteLine($"Error:
resource string \"{resourceStringName}\"

```

```

not found." + Environment.NewLine +
$"Search path:
{localizedString.SearchedLocation}");
ForegroundColor = previousColor; return
$"{localizedString}: "; } // 3. Return the
found resource string. return
localizedString; } public string?
GetEnterYourDobPrompt() { //
LocalizedString has an implicit cast to
string // that falls back to the key if
the resource // string is not found.
return localizer["EnterYourDob"]; } public
string? GetEnterYourSalaryPrompt() {
return localizer["EnterYourSalary"]; }
public string? GetPersonDetails( string
name, DateTime dob, int minutes, decimal
salary) { return
localizer["PersonDetails", name, dob,
minutes, salary]; } }

```

For the `GetEnterYourNamePrompt` method, I broke the implementation down into steps to get useful information, like checking if the resource string is found and showing the search path if not. The other method implementations use a simplified fallback to the key name for the resource string if the resource is not found.

1. In `Program.cs`, at the top, import the namespaces to work with hosting and dependency injection, and then configure a host that enables localization and the `PacktResources` service, as shown in the following code:

```
using Microsoft.Extensions.Hosting; // To
use IHost, Host. // To use
AddLocalization, AddTransient<T>. using
Microsoft.Extensions.DependencyInjection;
using IHost host =
Host.CreateDefaultBuilder(args)
.ConfigureServices(services => {
services.AddLocalization(options => {
options.ResourcesPath = "Resources"; });
services.AddTransient<PacktResources>();
}) .Build();
```

Good practice: By default, `ResourcesPath` is an empty string, meaning it looks for `.resx` files in the current directory. We are going to make the project tidier by putting resources into a subfolder.

1. After changing the current culture, add a statement to get the `PacktResources` service, and use it to output localized prompts for the user to enter their name, date of birth, and salary. Then, output their details, as highlighted in the following code:

```
OutputCultures("After changing the current
culture"); PacktResources resources =
host.Services.GetRequiredService<PacktResources>();
Write(resources.GetEnterYourNamePrompt());
string? name = ReadLine(); if
(string.IsNullOrEmpty(name)) { name =
"Bob"; }
Write(resources.GetEnterYourDobPrompt());
string? dobText = ReadLine(); if
```



```

(string.IsNullOrEmpty(dobText)) { //
    If they do not enter a DOB then use //
    sensible defaults for their culture.
    dobText = ci.Name switch { "en-US" or "fr-
    CA" => "1/27/1990", "da-DK" or "fr-FR" or
    "pl-PL" => "27/1/1990", "fa-IR" =>
    "1990/1/27", _ => "1/27/1990" }; }
Write(resources.GetEnterYourSalaryPrompt());
string? salaryText = ReadLine(); if
(string.IsNullOrEmpty(salaryText)) {
    salaryText = "34500"; } DateTime dob =
DateTime.Parse(dobText); int minutes =
(int)DateTime.Today.Subtract(dob).TotalMinutes;
decimal salary =
decimal.Parse(salaryText);
WriteLine(resources.GetPersonDetails(name,
dob, minutes, salary));

```

Testing globalization and localization

Now, we can run the console app and see the resources being loaded:

1. Run the console app and enter `da-DK` for the ISO code. Note that the prompts are in US English because we currently only have invariant culture resources.

To save time and to make sure you have the correct structure, you can copy, paste, and rename the `.resx` files instead of creating empty new ones. Or you can copy these files from the GitHub repository for the book.

1. In the `Resources` folder, add a new XML file named

PacktResources.da.resx, and modify the contents to contain non-region-specific Danish language resources, as shown in the following markup:

```
<?xml version="1.0" encoding="utf-8"?>
<root> <data name="EnterYourDob"
xml:space="preserve"> <value>Indtast din
fødselsdato: </value> </data> <data
name="EnterYourName" xml:space="preserve">
<value>Indtast dit navn: </value> </data>
<data name="EnterYourSalary"
xml:space="preserve"> <value>Indtast din
løn: </value> </data> <data
name="PersonDetails" xml:space="preserve">
<value>{0} blev født på en {1:dddd}. {0}
er {2:N0} minutter gammel. {0} tjener
{3:C}.</value> </data> </root>
```

2. In the Resources folder, add a new XML file named PacktResources.fr.resx, and modify the contents to contain non-region-specific French language resources, as shown in the following markup:

```
<?xml version="1.0" encoding="utf-8"?>
<root> <data name="EnterYourDob"
xml:space="preserve"> <value>Entrez votre
date de naissance: </value> </data> <data
name="EnterYourName" xml:space="preserve">
<value>Entrez votre nom: </value> </data>
<data name="EnterYourSalary"
xml:space="preserve"> <value>Entrez votre
salaire: </value> </data> <data
name="PersonDetails" xml:space="preserve">
```

```
<value>{0} est né un {1:dddd}. {0} a  
{2:N0} minutes. {0} gagne {3:C}.</value>  
</data> </root>
```

3. In the `Resources` folder, add a new XML file named `PacktResources.fr-CA.resx`, and modify the contents to contain the French language in Canada region resources, as shown in the following markup:

```
<?xml version="1.0" encoding="utf-8"?>  
<root> <data name="EnterYourDob"  
xml:space="preserve"> <value>Entrez votre  
date de naissance / Enter your date of  
birth: </value> </data> <data  
name="EnterYourName" xml:space="preserve">  
<value>Entrez votre nom / Enter your name:  
</value> </data> <data  
name="EnterYourSalary"  
xml:space="preserve"> <value>Entrez votre  
salaire / Enter your salary: </value> </  
data> <data name="PersonDetails"  
xml:space="preserve"> <value>{0} est né un  
{1:dddd}. {0} a {2:N0} minutes. {0} gagne  
{3:C}.</value> </data> </root>
```

4. In the `Resources` folder, add a new XML file named `PacktResources.pl-PL.resx`, and modify the contents to contain the Polish language in Poland region resources, as shown in the following markup:

```
<?xml version="1.0" encoding="utf-8"?>  
<root> <data name="EnterYourDob"  
xml:space="preserve"> <value>Wpisz swoją
```

```

datę urodzenia: </value> </data> <data
name="EnterYourName" xml:space="preserve">
<value>Wpisz swoje imię i nazwisko: </
value> </data> <data
name="EnterYourSalary"
xml:space="preserve"> <value>Wpisz swoje
wynagrodzenie: </value> </data> <data
name="PersonDetails" xml:space="preserve">
<value>{0} urodził się na {1:dddd}. {0} ma
{2:N0} minut. {0} zarabia {3:C}.</value>
</data> </root>

```

5. In the `Resources` folder, add a new XML file named `PacktResources_fa-IR.resx`, and modify the contents to contain the Farsi language in Iranian region resources, as shown in the following markup:

```

<?xml version="1.0" encoding="utf-8"?>
<root> <data name="EnterYourDob"
xml:space="preserve"> <value>تاریخ تولد خود را
وارد کنید / Enter your date of birth: </
value> </data> <data name="EnterYourName"
xml:space="preserve"> <value>اسمت را وارد کن
/ Enter your name: </value> </data> <data
name="EnterYourSalary"
xml:space="preserve"> <value>حقوق خود را وارد
کنید / Enter your salary: </value> </data>
<data name="PersonDetails"
xml:space="preserve"> <value>{0} در
1}:dddd}2} {0} به دنیا آمد. {0} {0} دقیقه است.
3}:C}.</value> </data> </root>

```

6. Run the code, and enter `da-DK` for the ISO code. Note that the prompts

are in Danish, as shown in the following output:

```
The current localization culture is da-DK:  
dansk (Danmark) ... Indtast dit navn: Bob  
Indtast din fødselsdato: 3/4/1987 Indtast  
din løn: 45449 Bob blev født på en fredag.  
Bob er 19.016.640 minutter gammel. Bob  
tjener 45.449,00 kr.
```

7. Run the code, and enter `fr-FR` for the ISO code. Note that the prompts are only in French, as shown in the following output:

```
The current localization culture is fr-FR:  
français (France) ... Entrez votre nom:  
Monique Entrez votre date de naissance:  
2/12/1990 Entrez votre salaire: 45000  
Monique est né un dimanche. Monique a 17  
088 480 minutes. Monique gagne 45 000,00  
€.
```

8. Run the code, and enter `fr-CA` for the ISO code. Note that the prompts are in French and English because Canada might have a requirement to support both as official languages, as shown in the following output:

```
The current localization culture is fr-CA:  
français (Canada) ... Entrez votre nom /  
Enter your name: Sophie Entrez votre date  
de naissance / Enter your date of birth:  
4/5/2001 Entrez votre salaire / Enter your  
salary: 65000 Sophie est né un jeudi.  
Sophie a 11 649 600 minutes. Sophie gagne  
65 000,00 $ CA.
```

9. Run the code, and enter `fa-IR` for the ISO code. Note that the prompts are in Persian/Farsi and English, and there is the additional complication of a right-to-left language, as shown in the following output:

```
The current localization culture is fa-IR:
اسمت را وارد کن ... / Enter your
name: Hoshyar تولد خود را وارد کنید / Enter
your date of birth: 1370/3/6 حقوق خود را وارد کنید / Enter your salary: 90000 Hoshyar در
دقیقه است Hoshyar 11,190,240. چهارشنبه دنیا آمد
Hoshyar 90,000ریال.
```

If you need to work with Persian dates, then there are NuGet packages with open-source GitHub repositories that you can try, although I cannot vouch for their correctness, like <https://github.com/VahidN/DNTPersianUtils.Core> and <https://github.com/imanabidi/PersianDate.NET>.

1. In the `Resources` folder, in `PacktResources.da.resx`, modify the contents to deliberately change the key for the prompt to enter your name, by appending `Wrong`, as highlighted in the following markup:

```
<?xml version="1.0" encoding="utf-8"?>
<root> <data name="EnterYourDob"
xml:space="preserve"> <value>Indtast din
fødselsdato: </value> </data> <data
name="EnterYourNameWrong"
xml:space="preserve"> <value>Indtast dit
```

```
navn: </value> </data> <data  
name="EnterYourSalary"  
xml:space="preserve"> <value>Indtast din  
løn: </value> </data> <data  
name="PersonDetails" xml:space="preserve">  
<value>{0} blev født på en {1:dddd}. {0}  
er {2:N0} minutter gammel. {0} tjener  
{3:C}.</value> </data> </root>
```

2. Run the code, and enter `da-DK` for the ISO code. Note that the prompts are in Danish, except for the `Enter your name` prompt in English, due to it falling back to the default resource file, as shown in the following output:

```
The current localization culture is da-DK:  
dansk (Danmark) ...  
Enter your name  
: Bob Indtast din fødselsdato: 3/4/1987  
Indtast din løn: 45449 Bob blev født på en  
fredag. Bob er 18.413.280 minutter gammel.  
Bob tjener 45.449,00 kr.
```

3. In the `Resources` folder, in `PacktResources.resx`, modify the contents to deliberately change the key for the prompt to enter your name, by appending `Wrong`.
4. Run the code, and enter `da-DK` for the ISO code. Note that the prompts are in Danish, except for the `Enter your name` prompt, which shows an error and uses the key name as a last-resort fallback, as shown in the following output:

```
The current localization culture is da-DK:  
dansk (Danmark) ...
```

Error: resource string "EnterYourName" not found.

Search path:

WorkingWithCultures.Resources.PacktResources
EnterYourName:

Bob Indtast din fødselsdato: 3/4/1987

Indtast din løn: 45449 Bob blev født på en fredag. Bob er 18.413.280 minutter gammel. Bob tjener 45.449,00 kr.

5. Remove the `Wrong` suffix in both resource files.
6. In **Solution Explorer**, toggle **Show All Files**, and expand the `bin/Debug/net10.0/da` folder, as shown in *Figure 6.1*:

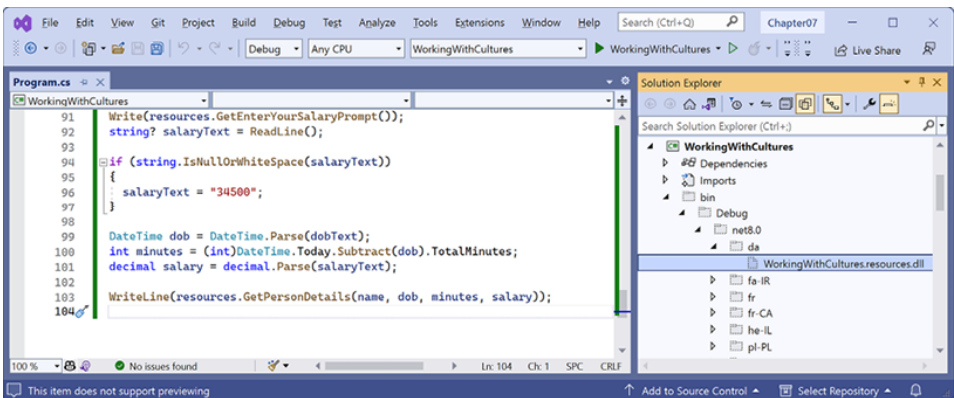


Figure 6.1: The satellite assembly folders for culture resources

1. Note the satellite assembly named `WorkingWithCultures.resources.dll` for the neutral Danish resources.

Any other culture resource assemblies are named the same but stored in folders that match the appropriate culture code. You can use tools like **ResX Resource Manager** (found at the following link: <https://dotnetfoundation.org/projects/resx-resource-manager>) to create many more `.resx` files, compile them into satellite assemblies, and then

deploy them to users without needing to recompile the original console app.

Good practice: Consider whether your application needs to be internationalized, and plan for that before you start coding! Think about all the data that will need to be globalized (date formats, number formats, and sorting text behavior). Write down all the pieces of text in the user interface that will need to be localized.

Microsoft has an online tool (found at the following link: <https://www.microsoft.com/en-us/Language/>) that can help you translate text in your user interfaces, as shown in *Figure 6.2*:

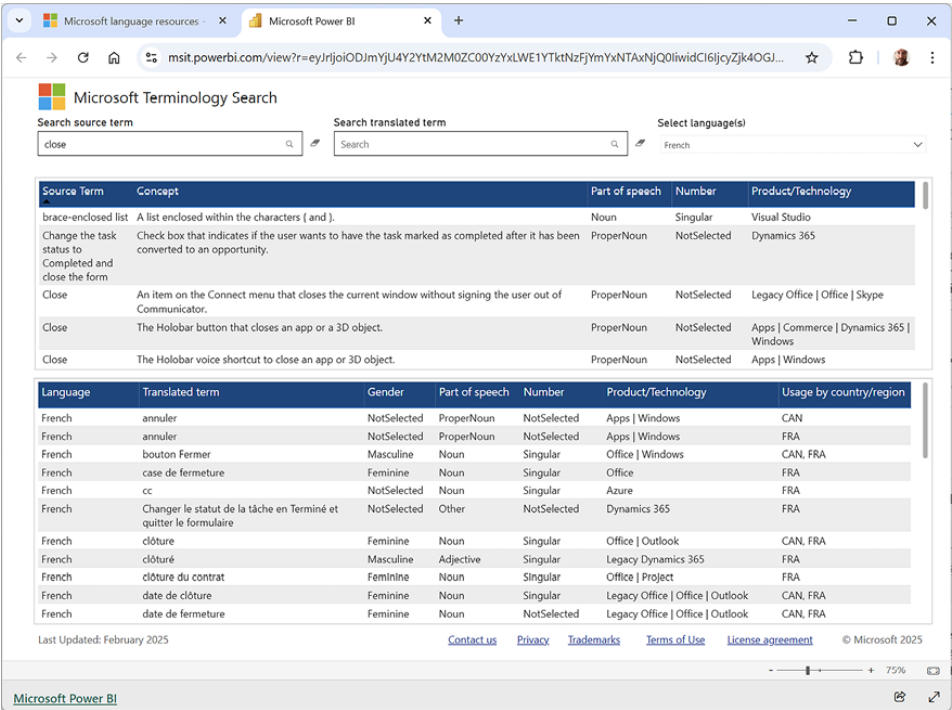


Figure 6.2: Microsoft user interface online text translation tool

We have now seen lots of date and time features provided by the .NET BCL.

Does it provide everything we need to handle internationalization?
Unfortunately, no. That's why you will likely want to use Noda Time.

Working with Noda Time

Noda Time is for developers who feel that the built-in libraries for handling dates and times are not good enough. Noda Time is like Joda Time, a replacement date/time handling library for Java.

Noda Time 3.0 or later supports .NET Standard 2.0 and .NET 6 or later. This means that you can use it with legacy platforms like .NET Framework and Xamarin, as well as modern .NET.

To understand one of the core deficiencies with the built-in .NET date/time types, imagine that instead of defining separate types for numbers, like `int` (`System.Int32`), `double` (`System.Double`), and `decimal` (`System.Decimal`), the .NET team defined only a `System.Number` type with a property named `Kind` to indicate what kind of number it is, how it is stored in memory, how to handle it, and so on.

That is what the team did with `System.DateTime`. That type has a `Kind` property that indicates if it is a local time, UTC time, or unspecified. It varies in behavior depending on how you treat it. This makes date/time values as implemented in .NET fundamentally tricky to work with and understand.

Jon Skeet, the creator of Noda Time, has a lot more to say about the limitations of date/time support in .NET and `DateTime`, specifically in a 2011 blog post found at the following link: <https://blog.nodatime.org/2011/08/what-wrong-with-datetime-anyway.html>.

Important concepts and defaults in Noda Time

The built-in `DateTime` type stores both global and local values, or values that are unspecified. Unless treated with great care, this opens the door to subtle bugs and misunderstandings.

The first big difference with Noda Time is that it forces you to make a choice at the type level. Noda Time, therefore, has more types and, at first, can seem more confusing. Types in Noda Time are global or local. Every person anywhere in the world would share the same global value at the same instant, whereas each person would have a different local value, depending on factors like their time zone.

The built-in date/time types in .NET are only accurate to the tick, which is about 100 nanoseconds. Noda Time is accurate to 1 nanosecond, 100 times more accurate.

The “zero” baseline in Noda Time is midnight at the start of 1 January 1970 in the UTC time zone. The Noda Time `Instant` is the number of nanoseconds before (if a negative value) or since (if a positive value) that time and represents a point in time on the global timeline.

The default calendaring system in Noda Time is the ISO-8601 calendar because it is the standard, and you can read more about it at the following link: https://en.wikipedia.org/wiki/ISO_8601. Automatic conversions to other calendaring systems like Julian, Coptic, and Buddhist are supported.

Noda Time has some common types that are similar to some .NET date/time types, as summarized in *Table 6.6*:

Noda Time type		
<code>Instant</code>	Represents an instant on the global timeline, with nanosecond resolution	
<code>Period</code>	Two <code>Instant</code> values, an inclusive start and an exclusive end	
<code>LocalDate</code>	Actual date values in a particular calendaring system, but it does not	

	represent a fixed point on the global timeline. For example, midnight on New Year’s Eve 2023 happens for different people in different time zones. If you do not know the user’s time zone, you will likely have to use this type		
	Like a <code>Date</code> and <code>Time</code> struct but only the date or time part. When prompting the user to enter date and time values, you often start with them separately and then combine them into a single <code>LocalDateTime</code> struct		
	Represents time zones. It is easy to convert from the .NET <code>TimeZoneInfo</code> using <code>BclDateTimeZone</code> . Use <code>DateTimeZoneProviders.Tzdb</code> to get a time zone, based on the standard names listed at the following link: https://en.wikipedia.org/wiki/List_of_tz_database_time_zones		
	A <code>DateTime</code> value in a particular calendaring system and in a specific time zone, so it does represent a fixed point on the global timeline		
	Represents an offset. It is positive if the local time is ahead of UTC, and negative if the local time is behind UTC		
	You might know the offset from UTC, but that does not always cleanly map to a single time zone. This type should be used in this scenario instead of <code>ZonedDateTime</code>		
	A fixed number of nanoseconds. Has properties to convert to common time units like <code>Days</code> , <code>Hours</code> , <code>Minutes</code> , and <code>Seconds</code> , rounded down or up to zero because they return an <code>int</code> , and non-rounded properties like <code>TotalDays</code> , <code>TotalMinutes</code> , and so on, because they return a <code>double</code> . Used for calculations on <code>Instant</code> and <code>ZonedDateTime</code> values. Use this instead of the .NET <code>TimeSpan</code> type		
	A variable duration because the “two months” represented by January and February 2024 are different lengths to the “two months” of June and July 2024, or even the “two months” represented by January and February in a non-leap year (2024 is a leap year, so February 2024 has 29 days).		

Table 6.6: Common Noda Time types

Good practice: Use `Instant` to record the point in time when something happened. It is good for timestamps. It can then be represented to the user in their local time zone. Common types used to record a user-entered date/time value are the following: `ZonedDateTime`, `OffsetDateTime`, `LocalDateTime`, `LocalDate`, and `LocalTime`.

Converting between Noda Time date/time types

To summarize common ways to convert between Noda Time types, review the following non-exhaustive diagram, see *Figure 6.3*:

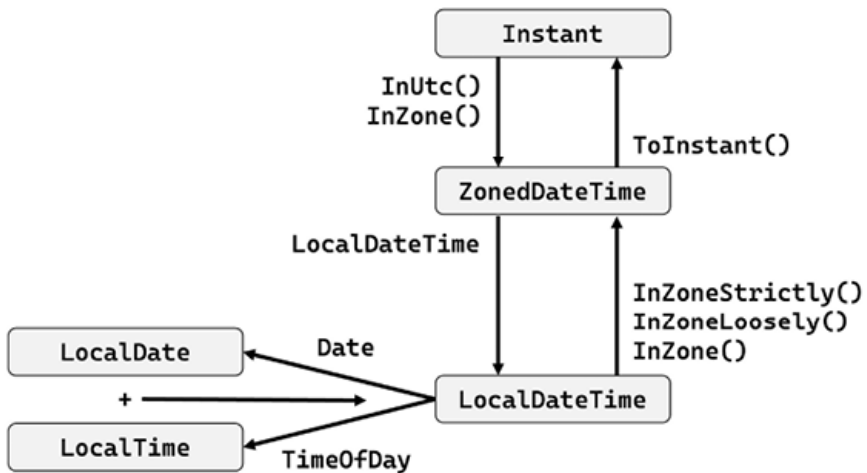


Figure 6.3: Common ways to convert between Noda Time date/time types

Exploring Noda Time in a console app

Let's write some code to explore some common ways to benefit from Noda Time, like getting an instance in time, converting a date/time value into UTC,

and working with time zones and periods:

1. Use your preferred code editor to add a new **Console App** / console project named `WorkingWithNodaTime` to the Internationalization solution.
2. In the project file, treat warnings as errors, then statically and globally import the `System.Console` class, and add a package reference for `NodaTime`, as shown in the following markup:

```
<ItemGroup> <PackageReference  
Include="NodaTime" /> </ItemGroup>
```

3. Add a new class file named `Program.Helpers.cs`, and replace its contents, as shown in the following code:

```
partial class Program { private static  
void SectionTitle(string title) {  
    ConsoleColor previousColor =  
        ConsoleColor; ConsoleColor =  
        ConsoleColor.DarkYellow; WriteLine($"***  
{title}"); ConsoleColor =  
        previousColor; } }
```

4. In `Program.cs`, delete the existing statements, add statements to get the current instant in time, and convert it to various Noda Time types, including UTC, a couple of time zones, and local time, as shown in the following code:

```
using NodaTime; // To use SystemClock,  
Instant and so on.  
SectionTitle("Converting Noda Time  
types"); // Get the current instant in
```

```
time. Instant now =
SystemClock.Instance.GetCurrentInstant();
WriteLine($"Now (Instant): {now}");
WriteLine(); ZonedDateTime nowInUtc =
now.InUtc(); WriteLine($"Now
(DateTimeZone): {nowInUtc.Zone}");
WriteLine($"Now (ZonedDateTime):
{nowInUtc}"); WriteLine($"Now (DST):
{nowInUtc.IsDaylightSavingTime()}");
WriteLine(); // Use the Tzdb provider to
get the time zone for US Pacific. // To
use .NET compatible time zones, use the
Bcl provider. DateTimeZone zonePT =
DateTimeZoneProviders.Tzdb["US/Pacific"];
ZonedDateTime nowInPT =
now.InZone(zonePT); WriteLine($"Now
(DateTimeZone): {nowInPT.Zone}");
WriteLine($"Now (ZonedDateTime):
{nowInPT}"); WriteLine($"Now (DST):
{nowInPT.IsDaylightSavingTime()}");
WriteLine(); DateTimeZone zoneUK =
DateTimeZoneProviders.Tzdb["Europe/
London"]; ZonedDateTime nowInUK =
now.InZone(zoneUK); WriteLine($"Now
(DateTimeZone): {nowInUK.Zone}");
WriteLine($"Now (ZonedDateTime):
{nowInUK}"); WriteLine($"Now (DST):
{nowInUK.IsDaylightSavingTime()}");
WriteLine(); LocalDateTime nowInLocal =
nowInUtc.LocalDateTime; WriteLine($"Now
(LocalDateTime): {nowInLocal}");
WriteLine($"Now (LocalDate):
{nowInLocal.Date}"); WriteLine($"Now
(LocalTime): {nowInLocal.TimeOfDay}");
WriteLine();
```

5. Run the console app, and note the results, including that “local” time does not take into account any DST offset; for example, in my case, living in the UK, I must use the London time zone to get British Summer Time (10:21am), not “local” time (9:21am), as shown in the following output:

```
*** Converting Noda Time types Now
(Instant): 2023-06-01T09:21:05Z Now
(DateTimeZone): UTC Now (ZonedDateTime):
2023-06-01T09:21:05 UTC (+00) Now (DST):
False Now (DateTimeZone): US/Pacific Now
(ZonedDateTime): 2023-06-01T02:21:05 US/
Pacific (-07) Now (DST): True Now
(DateTimeZone): Europe/London Now
(ZonedDateTime): 2023-06-01T10:21:05
Europe/London (+01) Now (DST): True Now
(LocalDateTime): 01/06/2023 09:21:05 Now
(LocalDate): 01 June 2023 Now (LocalTime):
09:21:05
```

6. In Program.cs, add statements to explore what can be done with periods of time, as shown in the following code:

```
SectionTitle("Working with periods"); //
The modern .NET era began with the release
of .NET Core 1.0 // on June 27, 2016 at
10am Pacific Time, or 5pm UTC.
LocalDateTime start = new(year: 2016,
month: 6, day: 27, hour: 17, minute: 0,
second: 0); LocalDateTime end =
LocalDateTime.FromDateTime(DateTime.UtcNow);
WriteLine("Modern .NET era");
WriteLine($"Start: {start}");
WriteLine($"End: {end}"); WriteLine();
```



```

Period period = Period.Between(start,
end); WriteLine($"Period: {period}");
WriteLine($"Years: {period.Years}");
WriteLine($"Months: {period.Months}");
WriteLine($"Weeks: {period.Weeks}");
WriteLine($"Days: {period.Days}");
WriteLine($"Hours: {period.Hours}");
WriteLine(); Period p1 =
Period.FromWeeks(2); Period p2 =
Period.FromDays(14); WriteLine($"p1
(period of two weeks): {p1}");
WriteLine($"p2 (period of 14 days):
{p2}"); WriteLine($"p1 == p2: {p1 ==
p2}"); WriteLine($"p1.Normalize() == p2:
{p1.Normalize() == p2}");

```

7. Run the console app and note the results, including that at the time of running the console app on 1 June 2023, the modern .NET era has lasted 6 years, 11 months, and 4 days, the serialization format for the `Period` type, and how two periods can be compared and should be normalized before the comparison, as shown in the following output:

```

*** Working with periods Modern .NET era
Start: 27/06/2016 17:00:00 End: 01/06/2023
09:21:05 Period:
P6Y11M4DT16H21M5S889s9240t Years: 6
Months: 11 Weeks: 0 Days: 4 Hours: 16 p1
(period of two weeks): P2W p2 (period of
14 days): P14D p1 == p2: False
p1.Normalize() == p2: True

```

Normalizing a `Period` with the `Normalize`

method means multiplying any weeks by 7 and adding them to the number of days, then setting Weeks to zero, and other calculations. Learn more at the following link: https://nodatime.org/3.1.x/api/NodaTime.Period.html#NodaTime_Period_NominalPeriod

[nodatime.org/3.1.x/api/](https://nodatime.org/3.1.x/api/NodaTime.Period.html#NodaTime_Period_NominalPeriod)

[NodaTime.Period.html#NodaTime_Period_NominalPeriod](https://nodatime.org/3.1.x/api/NodaTime.Period.html#NodaTime_Period_NominalPeriod)

Nanoseconds with Noda Time

Noda Time provides a much more robust and precise way to handle nanoseconds compared to the built-in .NET types.

As you've seen in the preceding code examples, Noda Time's core "point in time" type is `Instant`, which represents a global, unambiguous moment with nanosecond precision. One Noda Time "tick" is exactly one nanosecond.

Let's review some code that uses Noda Time to work with nanoseconds:

```
using NodaTime; using NodaTime.Text; using
System; // 1. Create a specific Instant with
nanosecond precision. // An Instant is ticks
from the Unix epoch (1970-01-01T00:00:00Z).
long nanosecondsSinceEpoch =
1_678_886_400_123_456_789L; Instant
preciseInstant =
Instant.FromUnixTimeNanoseconds(nanosecondsSinceEpoch)
// 2. We can access the nanosecond component
directly. // The output will be the
nanoseconds within the current second.
WriteLine($"Nanosecond of second:
{preciseInstant.NanosecondOfSecond}"); // 3.
Perform calculations with nanosecond
precision. // Let's add 50 nanoseconds.
Duration tinyDuration =
Duration.FromNanoseconds(50); Instant
```

```

newInstant = preciseInstant + tinyDuration; //
4. Format the output to show the full
precision. // 'fffffffffi' is the pattern for
nanoseconds. var pattern =
InstantPattern.CreateWithInvariantCulture(
"yyyy-MM-dd'T'HH:mm:ss.fffffffff'Z'");
WriteLine($"Original Instant:
{pattern.Format(preciseInstant)}");
WriteLine($"New Instant:
{pattern.Format(newInstant)}");

```

The output of the preceding code would be as follows:

```

Nanosecond of second: 123456789 Original
Instant: 2023-03-15T13:20:00.123456789Z New
Instant: 2023-03-15T13:20:00.123456839Z

```

While .NET 7 and later added `Microsecond` and `Nanosecond` properties to `DateTime`, Noda Time's entire design is built around precision and correctness, offering several key advantages, as shown in the following list:

- **True Nanosecond Precision.** The fundamental unit of Noda Time is the nanosecond. The built-in `DateTime` and `DateTimeOffset` types use a tick that is equal to 100 nanoseconds. Even in .NET 7 and later, the `Nanosecond` property is just a view over the underlying 100-nanosecond ticks. You can't represent a time with a ...001 nanosecond ending; it will always be a multiple of 100 (e.g., ...100, ...200). Noda Time has no such limitation.
- **Unambiguous and Explicit Types.** .NET's `DateTime` is famously problematic because its `Kind` property (`Utc`, `Local`, `Unspecified`) can create ambiguity. Is a `DateTime` value local to the server or the user? Noda Time avoids this by providing distinct types for different concepts:

- **Instant:** An absolute point in time on the UTC timeline. Always unambiguous. This is the equivalent of `DateTimeOffset` but with higher precision.
- **ZonedDateTime:** A time in a specific time zone (e.g., “Europe/London”), fully aware of DST rules.
- **LocalDateTime:** A date and time without a time zone (e.g., “December 25th at 8:00 AM”). It doesn’t represent a single point in time until you apply a time zone.
- **Superior Time Zone Handling.** .NET relies on the time zone information provided by the host operating system. Noda Time uses the industry-standard IANA Time Zone Database (TZDB), which it includes as part of the library. This provides richer, more accurate, and more consistent time zone information, regardless of the platform your code runs on.
- **Immutability.** All Noda Time types are immutable. When you perform an operation, like adding a `Duration`, it returns a new instance instead of modifying the existing one. This makes the code safer, especially in multi-threaded applications, and easier to reason about.

The documentation for Noda Time is found at the following link: <https://nodatime.org/>.

Practicing and exploring

Test your knowledge and understanding by answering some questions, getting some hands-on practice, and exploring the topics in this chapter with deeper research.

Exercise 6.1 – Online material

Online material can be extra content written by me for this book, or it can be references to content created by Microsoft or third parties.

Unit testing and JSON serialization with Noda Time

Noda Time has optional packages for special scenarios:

- To write unit tests: `NodaTime.Testing`
- To work with JSON.NET: `NodaTime.Serialization.JsonNet`

Falsehoods programmers believe about time

Noah Sussman's *Falsehoods Programmers Believe About Time* article is valuable for developers to read because it systematizes common misconceptions about temporal modeling in computational systems. It characterizes difficult cases such as temporal boundaries at day-and-year-end transitions, irregularities in calendar cycles, distributed clock drift, and jurisdiction-dependent variability in time zone definitions.

The result is a structured taxonomy of temporal edge cases designed to support more reliable and reproducible time-aware applications. You can find it at the following link:

<https://infinite-undo.medium.com/falsehoods-programmers-believe-about-time-0033203f0410>

Learn from expert Jon Skeet

Jon Skeet is a world-renowned expert on internationalization. Watch him present *Working with Time is Easy* at the following link: <https://www.youtube.com/watch?v=saeKBuPewcU>

Exercise 6.2 – Test your knowledge

Use the web to answer the following questions:

1. What is the difference between localization, globalization, and internationalization?

2. What is the smallest measurement of time available in .NET?
3. How long is a “tick” in .NET?
4. In what scenario might you use a `DateOnly` value instead of a `DateTime` value?
5. For a time zone, what does its `BaseUtcOffset` property tell you?
6. How can you get information about the local time zone in which your code executes?
7. For a `DateTime` value, what does its `Kind` property tell you?
8. How can you control the current culture for your executing code?
9. What is the ISO culture code for Welsh?
10. How do localization resource file fallbacks work?

Exercise 6.3 – Explore topics

Use the links on the following page to learn more details about the topics covered in this chapter: <https://github.com/markjprice/apps-services-net10/blob/main/docs/book-links.md#chapter-6---handling-dates-times-and-internationalization>.

Summary

In this chapter, you:

- Explored dates and times, including the `TimeProvider` to improve unit tests.
- Learned how to handle time zones.
- Learned how to internationalize your code using globalization and localization.
- Explored some of the features of Noda Time.

In the next chapter, you will learn how to use SQL Server to store and manage relational data.

Get This Book **UNLOCK NOW** and Exclusive

Scan the QR code
book by name, co



. Search for this
ps on the page.

***Note:** Keep your invoice handy. Purchases made directly from Packt don't require one.*

7

Managing Relational Data Using SQL

This chapter is about managing relational data stored in SQL Server hosted locally, Azure SQL Database, or SQL Server in a container like Docker. First, you will learn how to manage the data using native Transact-SQL statements. Next, you will learn how to manage data at a low level using ADO.NET libraries (`Microsoft.Data.SqlClient`). Finally, you will use Dapper to make it easier to work with entity models.

This chapter will cover the following topics:

- Understanding modern databases
- Managing data with Transact-SQL
- Managing SQL Server data with low-level APIs
- Managing SQL Server data with Dapper

Understanding modern databases

Two of the most common places to store data are in a **Relational Database Management System (RDBMS)**, such as **SQL Server**, **PostgreSQL**, **MySQL**, and **SQLite**, or in a **NoSQL** database, such as **Azure Cosmos DB**, **MongoDB**, **Redis**, and **Apache Cassandra**.

In this chapter, we will focus on the most popular RDBMS for Windows, which is SQL Server. This product is also available in a version for Linux. For cross-platform development, you can use either Azure SQL Database, which stores the data in the cloud, or SQL Server in a container, which can run in a Docker container on Windows, macOS, or Linux.

Using a sample relational database

To learn how to manage an RDBMS using .NET, it would be useful to have a sample one so that you can practice on a database that has a medium complexity and a decent number of sample records.

Microsoft offers several sample databases, most of which are too complex for our needs, so instead, we will use a database that was first created in the early 1990s, known as **Northwind**.

Let’s take a minute to look at a diagram of the Northwind database and its eight most important tables. You can refer to the diagram in *Figure 7.1* as we write code and queries throughout this book:



Figure 7.1: The Northwind database tables and relationships

Note that:

- Each category has a unique identifier, name, description, and picture. The picture is stored as a byte array in JPEG format.
- Each product has a unique identifier, name, unit price, number of units in stock, and other columns.
- Each product is associated with a category by storing the category's unique identifier.
- The relationship between `Categories` and `Products` is one-to-many, meaning each category can have zero, one, or more products. Each product must have a category.
- Each product is supplied by a supplier company indicated by storing the supplier's unique identifier.
- The quantity and unit price of a product are stored for each detail of an order.
- Each order is made by a customer, taken by an employee, and shipped by a shipping company.
- Each employee has a name, address, contact details, birth, and hire dates, a reference to their manager (except for the boss, whose `ReportsTo` field is `null`), and a photo stored as a byte array in JPEG format. The table has a one-to-many relationship with itself because one employee can manage many other employees.

Connecting to a SQL Server database

To connect to a SQL Server database, we need to know multiple pieces of information, as shown in the following list:

- The name of the server (and the name of the instance if it has more than the default one). This can include the protocol, IP address, and port number if connecting over a network.
- The name of the database.
- Security information, such as the username and password, or if we should

pass the currently logged-on user's credentials automatically using Windows Authentication.

We specify this information in a **connection string** by specifying multiple keyword-value pairs.

For backward compatibility, there are multiple possible keywords we can use in a SQL Server connection string for the various parameters, as shown in the following list:

- `Data Source`, `server`, or `addr`: These keywords are the name of the server (and an optional instance). You can use a dot (.) to mean the local server.
- `Initial Catalog` or `database`: These keywords are the name of the database that will be active initially. A SQL statement could change that using the command: `USE <databasename>`.
- `Integrated Security` or `trusted_connection`: These keywords are set to `true` or `SSPI` to pass the thread's current user credentials using Windows Authentication.
- `User Id` and `Password`: These keywords are used to authenticate with any edition of SQL Server. This is important for Azure SQL Database or SQL Server in a container because they do not support Windows Authentication. The full edition of SQL Server on Windows supports both username with password and Windows Authentication.
- `Authentication`: This keyword is used to authenticate by using Azure AD identities that can enable password-less authentication. Values can be `Active Directory Integrated`, `Active Directory Password`, and `Sql Password`.
- `Persist Security Info`: If set to `false`, this keyword tells the connection to remove the `Password` from the connection string after authenticating.
- `Encrypt`: If set to `true`, this keyword tells the connections to use SSL to encrypt transmissions between client and server.

- `TrustServerCertificate`: Set to `true` if hosting locally, and you get the error “A connection was successfully established with the server, but then an error occurred during the login process. (provider: SSL Provider, error: 0 - The certificate chain was issued by an authority that is not trusted.)”.
- `Connection Timeout`: This keyword defaults to 30 seconds.
- `MultipleActiveResultSets`: This keyword is set to `true` to enable a single connection to be used to work with multiple tables simultaneously to improve efficiency. It is used for lazy loading rows from related tables.

As described in the preceding list, when you write code to connect to a SQL Server database, you need to know its server name.

The server name depends on the edition and version of SQL Server that you will connect to, as shown in *Table 7.1*:

SQL Server edition	Instance name		
LocalDB (2016 or later)	localdb		
Express	Express		
Full/Developer (default instance)			
Full/Developer (named instance)			
SQL Server in a container (local Docker)			
Azure SQL Database			
	server name	.database.windows.net,	1433

Table 7.1: Server name examples for various editions of SQL Server

Good practice: Use a dot (.) as shorthand for the local computer name (localhost). Remember that server names for SQL Server can be made up of two parts: the name of the computer and the name of a SQL Server instance. You provide instance names during custom installation.

Now that you've reviewed how to connect to a SQL Server instance, let's review how to manage a database using Transact-SQL, the native language of SQL Server.

Managing data with Transact-SQL

Transact-SQL (T-SQL) is SQL Server's dialect of **Structured Query Language (SQL)**. Some pronounce it *tee-sequel*, others *tee-es-queue-el*.

Unlike C#, T-SQL is not case-sensitive; for example, you can use `int` or `INT` to specify the 32-bit integer data type, and you can use `SELECT` or `select` to start a query expression. Text data stored in SQL Server tables can be treated as case-sensitive or not, depending on the configuration.

T-SQL is based on the ANSI SQL standard, but it extends it with additional keywords and features specific to SQL Server. These extensions allow you to write procedural logic such as loops and conditionals, manage transactions, handle errors, and work with variables. In other words, T-SQL turns SQL from a purely declarative language ("what data do you want?") into a language that can also describe how to process that data on the server.

You use T-SQL whenever you communicate with a SQL Server database: in ad-hoc queries, stored procedures, views, triggers, and user-defined functions. Application code (for example, C# programs using ADO.NET or EF Core) ultimately sends T-SQL statements to the database engine for execution.

T-SQL code executes directly on the database server, close to the data itself. This allows for more efficient data processing and reduces the amount of data transferred over the network. It also gives database administrators fine-grained control over permissions and transactions.

In the sections that follow, you'll explore the core data types, expressions, and statements used to manage and query data with T-SQL.

The complete reference for T-SQL is found at the following link: <https://>

learn.microsoft.com/en-us/sql/t-sql/language-reference. From that documentation starting page, use the left side navigation to view topics like **Data types**, **Queries**, and **Statements**.

T-SQL data types

T-SQL has data types that are used for columns, variables, parameters, and so on, as shown in *Table 7.2*:

Examples		
Numbers	bit, decimal, float, int, money, numeric, real, smallint, smallmoney, tinyint	
Date and time	datetime2, datetime, datetimeoffset, smalldatetime, time	
Text	nchar, ntext, nvarchar, text, varchar	
Binary	image, varbinary	
GUID	uniqueidentifier (128-bit value typically displayed as a 36-character string such as: 6F9619FF-8B86-D011-B42D-00C04FC964FF)	
Other	hierarchyid, sql_variant, table, rowversion, xml	

Table 7.2: Categories of SQL Server data types

In SQL Server 2022 and earlier, there is an `xml` data type, but no JSON data type. If using one of those versions, use `nvarchar` to store JSON values. However from SQL Server 2025, JSON types have now been introduced.

T-SQL also has support for spatial `geometry` and `geography` types.

Data types for the primary key

When choosing the primary key data type for a SQL relational database, the decision often depends on your application's requirements, scalability considerations, and the way you intend to use the data. The data types that you choose for the primary key aren't different from the data types that you might pick for a column in a table. But you can't just use any data type for primary keys since they will be indexed and must have unique values. For example, you wouldn't choose `image` for the data type of a primary key!

The most common data types used for the primary key and their pros and cons include the following:

- **Integers:** `int` typically uses 4 bytes (32-bit) or 8 bytes (`bigint`, 64-bit), making it space-efficient. Numeric comparisons and indexing are faster than text or GUID values. Database operations (joins, lookups, and sorting) are efficient due to the compact size. Sequential numbers are easy for humans to understand and debug, but have a limited range (for example, a signed `int` maxes out at ~2.1 billion). Large-scale applications might outgrow this limit, though `bigint` can mitigate this. And if your database needs to support distributed systems, for example, microservices or sharded databases, generating unique integers across systems can be tricky.
- **GUIDs:** GUID values are 128-bit values designed to be unique across space and time, which is useful in distributed systems or multi-server environments. They eliminate the risk of collisions when merging data from different systems. GUID values can be generated by the application without needing a round trip to the database. They are suitable for scenarios where primary key generation happens offline or on the client side. There is no practical limit on how many GUID values you can generate. But a GUID value is 16 bytes (128 bits), which is four times larger than an `int` value. This increases storage requirements and can slow down indexing and lookups, so comparisons and indexing are slower due to the larger size. GUID values, particularly random ones, can

lead to fragmentation in clustered indexes, degrading performance. GUID values are long and cryptic, for example, 9A6E6F15-4F99-4C94-89FB-30DB0D3F39AB, making debugging more challenging.

Primary key data type choices

When deciding between integer and GUID types for a primary key, consider the following:

1. Application Scale and Usage:

- For small or medium-scale applications with a single database, `INT` (or `BIGINT`) is often sufficient.
- For distributed or multi-database systems, `GUID` may be necessary to ensure global uniqueness.

4. Performance, Storage and Indexing:

- If performance is critical, avoid `GUID` values as they can bloat indexes and slow down queries.
- If the table is large or frequently queried, the smaller storage footprint of `int` helps keep indexes compact and fast.

Good practice: Prefer `int` for most applications due to its simplicity, performance, and compactness. Use `bigint` if scalability beyond 2 billion rows is a concern. Consider `guid` if you need global uniqueness across distributed systems, or you anticipate frequent data merges from multiple systems. Each choice has its place, and the “best” option ultimately depends on your specific use case and priorities.

Documenting with comments

To comment out the rest of a line, use `--`, which is the equivalent of `//`.

To comment out a block, use `/*` at the start and `*/` at the end, just like in C#.

Declaring variables

Local variable names are prefixed with `@` and are defined using `SET`, `SELECT`, or `DECLARE`, as shown in the following code:

```
DECLARE @WholeNumber INT; -- Declare a
variable and specify its type. SET
@WholeNumber = 3; -- Set the variable to a
literal value. SET @WholeNumber = @WholeNumber
+ 1; -- Increment the variable. SELECT
@WholeNumber = COUNT(*) FROM Employees; -- Set
to the number of employees. SELECT
@WholeNumber = EmployeeId FROM Employees WHERE
FirstName = 'Janet';
```

Global variables are prefixed with `@@`. For example, `@@ROWCOUNT` is a context-dependent value that returns the number of rows affected by a statement executed within the current scope, for example, the number of rows updated or deleted.

Specifying data types

Most types have a fixed size. For example, an `int` uses four bytes, a `smallint` uses two bytes, and a `tinyint` uses one byte.

For text and binary types, you can either specify a type prefixed with `var` or `nvar` (meaning variable size), which will automatically change its size based on its current value up to a maximum, as shown in the following example:

`varchar(40)`; or you can specify a fixed number of characters that will

always be allocated, as shown in the following example: `char(40)`.

For text types, the `n` prefix indicates Unicode, meaning they will use two bytes per character. Text types not prefixed with `n` use one byte per character.

Controlling flow

T-SQL has similar flow control keywords as C#, for example, `BREAK`, `CONTINUE`, `GOTO`, `IF...ELSE`, `CASE`, `THROW`, `TRY...CATCH`, `WHILE`, and `RETURN`. The main difference is the use of `BEGIN` and `END` to indicate the start and end of a block, the equivalent of curly braces in C#.

Operators

T-SQL has similar operators as C#, for example, `=` (assignment), `+`, `-`, `*`, `/`, `%`, `<`, `>`, `<=`, `==`, `!=`, `&`, `|`, `^`, `IS NULL`, and so on. It has logical operators like `AND`, `OR`, `NOT`, and LINQ-like operators like `ANY`, `ALL`, `SOME`, `EXISTS`, `BETWEEN`, and `IN`.

Warning! Because `NULL` represents “unknown” or “missing” data in SQL, you can’t test for it with the usual equality operator (`=`). Instead, you must use `IS NULL` or `IS NOT NULL`.

`LIKE` is used for text pattern matching. The pattern can use `%` for any number of characters. The pattern can use `_` for a single character. The pattern can use `[]` to specify a range and set of allowed characters, for example, `[0-9A-Z.-,]`, which looks like a simplified regular expression syntax, but keep in mind that it is *not* regular expression syntax.

If a table or column name contains spaces, then you must surround the name in square brackets, like `[Order Details]`. The SQL scripts to create the Northwind database include the command `set`

quoted_identifier on, so you can also use double quotes, like "Order Details". Single quotes are used for literal text, like 'USA'.

Data Manipulation Language (DML)

DML is used to query and change data.

The most common statement in DML is SELECT, which is used to retrieve data from one or more tables. SELECT is extremely complicated because it is so powerful. This book is not about learning T-SQL, so the quickest way to get a feel for SELECT is to see some examples, as shown in *Table 7.3*:

Description		
SELECT columns of all the employees. FROM Employees		
Get the first and last name columns of all employees. FROM Employees		
Give an alias for the table name. Table name prefixes are not needed when there is only one table. As they become useful to disambiguate when there are multiple tables that have columns with the same name, for example, Customers.CustomerId and Orders.CustomerId		
Give an alias for the table name without needing the AS keyword. FROM Employees emp		
Give an alias for the column name AS Surname FROM Employees		
Filter the results to only include employees in the USA. FROM Employees WHERE Country = 'USA'		
SELECT list of columns used as values in the Country column of the Employees table without duplicates		
Calculate a subtotal for each order detail row. Subtotal FROM [Order Details]		

Calculate a total for each order and sort with the largest order value at the top <pre> SELECT OrderId, SUM(UnitPrice * Quantity) AS Total FROM [Order Details] GROUP BY OrderId ORDER BY Total DESC </pre>		
Return all the company names of all customers and suppliers. <pre> FROM Customers UNION SELECT CompanyName FROM Suppliers </pre>		
Match every category with every product using a Cartesian join and output the results (not what you normally want!). 616 rows (8 categories x 77 products)		
Match each product with its category using a WHERE clause for the foreign key column in each table and output the category name and product name <pre> WHERE c.CategoryId = p.CategoryId </pre> 77 rows		
Match each product with its category using an INNER JOIN...ON clause for the foreign key column in each table and output the category name and product name. This is a modern alternative syntax to using .WHERE and it allows outer joins which would also include non-matches. 77 rows.		

Table 7.3: Example SELECT statements with descriptions

You can read the full documentation for `SELECT` at the following link: <https://learn.microsoft.com/en-us/sql/t-sql/queries/select-transact-sql>.

Use your favorite database querying tool, like Visual Studio’s **Server Explorer** or VS Code’s `mssql` extension, to connect to your Northwind database and try

out some of the queries above, as shown in *Figure 7.2* and *Figure 7.3*:

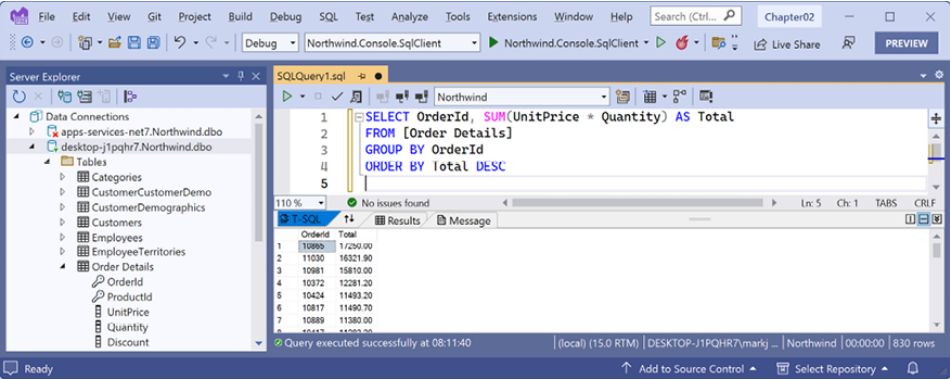


Figure 7.2: Executing T-SQL queries using Visual Studio’s Server Explorer

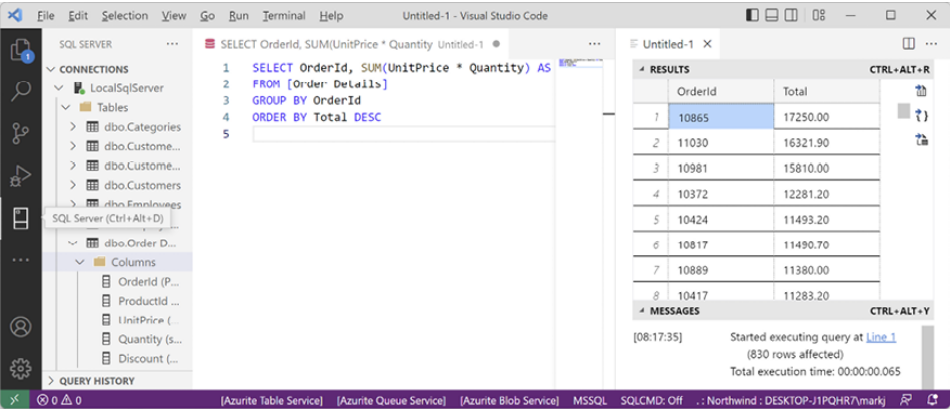


Figure 7.3: Executing T-SQL queries using VS Code’s mssql extension

DML for adding, updating, and deleting data

DML statements for adding, updating, and deleting data include those shown in *Table 7.4*:

Description		
Add a new row to the employees table (the name, last name, and phone number are automatically assigned. Use @@IDENTITY to get this value		

Update my employees	Update my employees to set my Country to UK.
SET Country = 'UK'	
WHERE FirstName = 'Mark'	
AND LastName = 'Price'	
Delete my employees.	
WHERE FirstName = 'Mark'	
AND LastName = 'Price'	
Delete all rows in the employees table and record those deletions in the transaction log	
Delete all rows in the employees table more efficiently because it does not log the individual row deletions.	

Table 7.4: Example DML statements with descriptions

The above examples use the `Employees` table in the Northwind database. That table has referential integrity constraints, which means that, for example, deleting all rows in the table cannot happen because every employee has related data in other tables like `Orders`.

Data Definition Language (DDL)

DDL statements change the structure of the database, including creating new objects like tables, functions, and stored procedures. The following table shows some examples of DDL statements to give you an idea, but the examples are simple and cannot be executed within the Northwind database, as shown in Table 7.5:

Description	
Create table to store shippers	
ShipperId INT PRIMARY KEY CLUSTERED,	
CompanyName NVARCHAR(40)	

You can find the GitHub repository for ADO.NET at the following link: <https://github.com/dotnet/SqlClient>.

The `Microsoft.Data.SqlClient` package supports the following .NET platforms:

- .NET Framework 4.6.2 and later.
- .NET Core 3.1 and later.
- .NET Standard 2.0 and later.

Warning! All

`System.Data.SqlClient` users are encouraged to transition to `Microsoft.Data.SqlClient`. You can read the announcement about how `System.Data.SqlClient` package is now deprecated at the following link: <https://techcommunity.microsoft.com/t5/sql-server-blog/announcement-system-data-sqlclient-package-is-now-deprecated/ba-p/4227205>.

Understanding the types in ADO.NET

ADO.NET defines abstract types that represent minimal objects for working with data, like `DbConnection`, `DbCommand`, and `DbDataReader`. Database software manufacturers can inherit from these abstract types and then provide specific implementations for their database system. These implementations are optimized for, and can expose, additional features.

Microsoft has done this for SQL Server. The most important types with their most used members are shown in *Table 7.6*:

Description							
Change the connection to the database.							
CreateCommand,							
ServerVersionStatistics							
Build a valid connection string for SQL Server database ID,							
Absorbing all the relevant individual properties, get the							
ConnectionString property							
Configure the command to execute							
ExecuteNonQuery,							
ExecuteXmlReader,							
ExecuteParameter							
Transaction							
SqlDataAdapter							
SetSqlAdapter for a command.							
Value, DbType,							
SqlValue,							
SqlDbType,							
Direction,							
IsNullable							
Repeatable set from executing a query.							
HasColumn,							
GetInfo2, GetString,							
RetDataAffected							
GetFieldValue<T>							

Table 7.6: Important types in ADO.NET SqlClient

SqlConnection has two useful events: `StateChange` and `InfoMessage`.

All the `ExecuteXxx` methods of `SqlCommand` will execute any command. The one you use depends on what you expect to get back:

- If the command includes at least one `SELECT` statement that returns a result set, then call `ExecuteReader` to execute the command. This method returns a `DbDataReader`-derived object for reading row-by-row through the result set.
- If the command does not include at least one `SELECT` statement, then it is more efficient to call `ExecuteNonQuery`. This method returns an integer for the number of rows affected.
- If the command includes at least one `SELECT` statement that returns XML because it uses the `AS XML` command, then call `ExecuteXmlReader` to execute the command.

Creating a console app for working with ADO.NET

First, we will create a console app project for working with ADO.NET:

1. Use your preferred code editor to create a console app project, as defined in the following list:
 - Project template: **Console App** / `console`
 - Solution file and folder: `ModernApps`
 - Project file and folder:
`Northwind.Console.SqlClient`
 - **Do not use top-level statements**: Cleared
 - **Enable native AOT publish**: Cleared
7. In the project file, treat warnings as errors, add a package reference for the latest version of `Microsoft.Data.SqlClient`, and statically and globally import `System.Console`, as shown highlighted in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup> <OutputType>Exe</
  OutputType> <TargetFramework>net10.0</
```

```

TargetFramework> <ImplicitUsings>enable</
ImplicitUsings> <Nullable>enable</
Nullable> <TreatWarningsAsErrors>true</
TreatWarningsAsErrors> </PropertyGroup>
<ItemGroup> <PackageReference
Include="Microsoft.Data.SqlClient" /> </
ItemGroup> <ItemGroup> <Using
Include="System.Console" Static="true" />
</ItemGroup> </Project>

```

8. Build the project to restore the referenced package.
9. Add a new class file named `Program.Helpers.cs`, and modify its contents to define a method to configure the console to enable special characters like the Euro currency symbol and set the current culture, and a method that will output some text to the console in a specified color, with a default color of black, as shown in the following code:

```

using System.Globalization; // To use
CultureInfo. partial class Program {
private static void
ConfigureConsole(string culture = "en-US",
bool useComputerCulture = false) { // To
enable Unicode characters like Euro symbol
in the console. OutputEncoding =
System.Text.Encoding.UTF8; if (!
useComputerCulture) {
CultureInfo.CurrentCulture =
CultureInfo.GetCultureInfo(culture); }
WriteLine($"CurrentCulture: {
CultureInfo.CurrentCulture.DisplayName}");
} private static void
WriteLineInColor(string value,
ConsoleColor color = ConsoleColor.White) {
ConsoleColor previousColor =

```

```
ForegroundColor; ForegroundColor = color;  
WriteLine(value); ForegroundColor =  
previousColor; } }
```

The default foreground color in the preceding code is white because I have assumed that most readers will have a default background color of black. On my computer, I set the default background color of the console to white so that I can take screenshots for this book. Set whatever default color is best for your computer.

1. Add a new class file named `Program.EventHandlers.cs`, and modify its contents to define methods that act as event handlers for a database connection state change by showing the original and current states, and for when the database sends an `InfoMessage`, as shown in the following code:

```
using Microsoft.Data.SqlClient; // To use  
SqlInfoMessageEventArgs. using  
System.Data; // To use  
StateChangeEventArgs. partial class  
Program { private static void  
Connection_StateChange( object sender,  
StateChangeEventArgs e) {  
WriteLineInColor( $"State change from  
{e.OriginalState} to {e.CurrentState}.",  
ConsoleColor.DarkYellow); } private static  
void Connection_InfoMessage( object  
sender, SqlInfoMessageEventArgs e) {  
WriteLineInColor($"Info: {e.Message}.",  
ConsoleColor.DarkBlue); } }
```

2. In `Program.cs`, delete the existing statements. Add statements to connect to SQL Server locally, to Azure SQL Database, or to SQL Server in a container, using either SQL authentication with a user ID and password or Windows Authentication without a user ID and password, as shown in the following code:

The following code is long but easy to understand. This is a good opportunity to copy blocks of code from the file in the GitHub repository instead of typing it all yourself: <https://github.com/markjprice/apps-services-net10/blob/main/code/ModernApps/Northwind.Console.SqlClient/Program.cs>.

```
using Microsoft.Data.SqlClient; // To use
SqlConnection and so on. ConfigureConsole();
#region Set up the connection string builder
SqlConnectionStringBuilder builder = new() {
    InitialCatalog = "Northwind",
    MultipleActiveResultSets = true, Encrypt =
    true, TrustServerCertificate = true,
    ConnectTimeout = 10 // Default is 30 seconds.
}; WriteLine("Connect to:"); WriteLine(" 1 -
SQL Server on local machine"); WriteLine(" 2 -
Azure SQL Database"); WriteLine(" 3 - SQL
Server in a container"); WriteLine();
Write("Press a key: "); ConsoleKey key =
ReadKey().Key; WriteLine(); WriteLine();
switch (key) { case ConsoleKey.D1 or
ConsoleKey.NumPad1: builder.DataSource = ".";
```

```

break; case ConsoleKey.D2 or
ConsoleKey.NumPad2: builder.DataSource = //
Use your Azure SQL Database server name.
"tcp:apps-services-
book.database.windows.net,1433"; break; case
ConsoleKey.D3 or ConsoleKey.NumPad3:
builder.DataSource = "tcp:127.0.0.1,1433";
break; default: WriteLine("No data source
selected."); return; } WriteLine("Authenticate
using:"); WriteLine(" 1 - Windows Integrated
Security"); WriteLine(" 2 - SQL Login, for
example, sa"); WriteLine(); Write("Press a
key: "); key = ReadKey().Key; WriteLine();
WriteLine(); if (key is ConsoleKey.D1 or
ConsoleKey.NumPad1) {
builder.IntegratedSecurity = true; } else if
(key is ConsoleKey.D2 or ConsoleKey.NumPad2) {
Write("Enter your SQL Server user ID: ");
string? userId = ReadLine(); if
(string.IsNullOrEmpty(userId)) {
WriteLine("User ID cannot be empty or null.");
return; } builder.UserID = userId;
Write("Enter your SQL Server password: ");
string? password = ReadLine(); if
(string.IsNullOrEmpty(password)) {
WriteLine("Password cannot be empty or
null."); return; } builder.Password =
password; builder.PersistSecurityInfo = false;
} else { WriteLine("No authentication
selected."); return; } #endregion #region
Create and open the connection // Connection
will Dispose/Close at the end of its scope.
using SqlConnection connection =
new(builder.ConnectionString);
WriteLine(connection.ConnectionString);

```

```

WriteLine(); connection.StateChange +=
Connection_StateChange; connection.InfoMessage
+= Connection_InfoMessage; try {
WriteLine("Opening connection. Please wait up
to {0} seconds...", builder.ConnectTimeout);
WriteLine(); connection.Open();
WriteLine($"SQL Server version:
{connection.ServerVersion}"); } catch
(SqlException ex) { WriteLineInColor($"SQL
exception: {ex.Message}", ConsoleColor.Red);
return; } #endregion

```

Good practice: In this coding task, we prompt the user to enter the password to connect to the database. In a real-world app, you are more likely to store the password in an environment variable or secure storage like Azure Key Vault. You must *never* store passwords in your source code!

1. Run the console app, select options that work with your SQL Server setup, and note the results, including the state change event output written in dark yellow to make them easier to see, as shown in the following output:

```

Connect to: 1 - SQL Server on local
machine 2 - Azure SQL Database 3 - SQL
Server in a container Press a key: 1
Authenticate using: 1 - Windows Integrated
Security 2 - SQL Login, for example, sa
Press a key: 1 Data Source=.;Initial
Catalog=Northwind;Integrated

```

```
Security=True;Multiple Active Result  
Sets=True;Connect  
Timeout=10;Encrypt=True;Trust Server  
Certificate=True Opening connection.  
Please wait up to 10 seconds... State  
change from Closed to Open. SQL Server  
version: 15.00.2101 State change from Open  
to Closed.
```

The following steps show the experience when connecting to Azure SQL Database or SQL Server in a container, which both require a username and password. If you are connecting to a local SQL Server using Windows Integrated Security, then you will not need to enter a password.

1. Run the console app, select choices that require a user ID like `sa` and a password like `s3cret-Ninja`, for example, with Azure SQL Database or SQL Server in a container, and note the result, as shown in the following output:

```
Enter your SQL Server user ID: sa Enter  
your SQL Server password: [censored] Data  
Source=tcp:127.0.0.1,1433;Initial  
Catalog=Northwind;Persist Security  
Info=False;User  
ID=sa;Password=[censored];Multiple Active  
Result Sets=True;Connect  
Timeout=10;Encrypt=True;Trust Server  
Certificate=True Opening connection.  
Please wait up to 10 seconds... State  
change from Closed to Open. SQL Server
```



```
version: 17.00.0800 State change from Open  
to Closed.
```

2. Run the console app, select choices that require a user ID and password, enter a wrong password, and note the result, as shown in the following output:

```
Enter your SQL Server user ID: sa Enter  
your SQL Server password: 123456 Data  
Source=tcp:127.0.0.1,1433;Initial  
Catalog=Northwind;Persist Security  
Info=False;User  
ID=sa;Password=123456;Multiple Active  
Result Sets=True;Connect  
Timeout=10;Encrypt=True;Trust Server  
Certificate=True Opening connection.  
Please wait up to 10 seconds... SQL  
exception: Login failed for user 'sa'.
```

3. In `Program.cs`, change the server name (the `DataSource` property) to something wrong.
4. Run the console app and note the result (depending on where your database is hosted, the exception message might be slightly different), as shown in the following output:

```
SQL exception: A network-related or  
instance-specific error occurred while  
establishing a connection to SQL Server.  
The server was not found or was not  
accessible. Verify that the instance name  
is correct and that SQL Server is  
configured to allow remote connections.  
(provider: TCP Provider, error: 0 - No
```

```
such host is known.)
```

When opening a SQL Server connection, the default timeout is 30 seconds for server connection problems, so be patient! We changed the timeout to 10 seconds to avoid having to wait so long.

Executing queries and working with data readers using ADO.NET

Now that we have a successful connection to the SQL Server database, we can run commands that retrieve rows from a table and process the results using a data reader:

1. In `Program.cs`, import the namespace for working with ADO.NET command types, as shown in the following code:

```
using System.Data; // To use CommandType.
```

Good practice: To save space in this book, I will use the names `cmd` and `r` to represent an SQL command and an SQL data reader. In your code, give variables proper word names like `command` and `reader`.

1. Before the statement that closes the connection, add statements to define a command that selects the ID, name, and price from the `Products` table, executes it, and outputs the product IDs, names, and prices using a data reader, as shown in the following code:

```
using SqlCommand cmd =
```

```

connection.CreateCommand();
cmd.CommandType = CommandType.Text;
cmd.CommandText = "SELECT ProductId,
ProductName, UnitPrice FROM Products";
using SqlDataReader r =
cmd.ExecuteReader(); string horizontalLine
= new string('-', 60);
WriteLine(horizontalLine); WriteLine("|
{0,5} | {1,-35} | {2,10} |", arg0: "Id",
arg1: "Name", arg2: "Price");
WriteLine(horizontalLine); while
(r.Read()) { WriteLine("| {0,5} | {1,-35}
| {2,10:C} |", r.GetInt32("ProductId"),
r.GetString("ProductName"),
r.GetDecimal("UnitPrice")); }
WriteLine(horizontalLine); r.Close();

```

We format the unit price using the `C` format, which uses the current culture to format currency values. The call to `ConfigureConsole` sets the current culture to US English, so the output for all readers uses `$`. To test alternative cultures like French that use the Euro currency symbol, modify the call at the top of the `Program.cs` file, as shown in the following code:

```
ConfigureConsole("fr-FR");
```

1. Run the console app and note the results, as shown in the following partial output:

Id	Name	Price
1	Chai	\$18.00
2	Chang	\$19.00
...	76	Lakkalikööri
		\$18.00
	77	Original Frankfurter grüne Soße
		\$13.00

2. In `Program.cs`, modify the SQL statement to define a parameter for the unit price and use it to filter the results to products that cost more than that unit price, as shown highlighted in the following code:

```
Write("Enter a unit price: "); string?
priceText = ReadLine(); if(!
decimal.TryParse(priceText, out decimal
price)) { WriteLine("You must enter a
valid unit price."); return; } using
SqlCommand cmd =
connection.CreateCommand();
cmd.CommandType = CommandType.Text;
cmd.CommandText = "SELECT ProductId,
ProductName, UnitPrice FROM Products" + "
WHERE UnitPrice >= @minimumPrice";
cmd.Parameters.AddWithValue("minimumPrice",
price);
```

3. Run the console app, enter a unit price like 50, and note the results, as shown in the following partial output:

Enter a unit price: 50

Id	Name	Price
----	------	-------

```
| 9 | Mishi Kobe Niku | $97.00 | | 18 |
Carnarvon Tigers | $62.50 | | 20 | Sir
Rodney's Marmalade | $81.00 | | 29 |
Thüringer Rostbratwurst | $123.79 | | 38 |
Côte de Blaye | $263.50 | | 51 | Manjimup
Dried Apples | $53.00 | | 59 | Raclette
Courdavault | $55.00 |
```

Outputting statistics

An ADO.NET connection can track useful statistics during its lifetime, including those listed in *Table 7.7*:

Description		
Bytes transferred as bytes stored in buffers, bytes received, BytesSent		
Cursors are an expensive operation because they require state on the server, and should be avoided when possible		
Number of prepared (compilations), executions of prepared commands, and executions of unprepared commands		
Number of SELECT statements and rows returned by SELECT statements		
Number of server round trips, requests, and transactions		
Time in milliseconds spent connected, executing commands, or the cumulative amount of time that the provider spent waiting for replies from the server once the application has started.		

Table 7.7: Connection statistics that can be tracked

Let's enable this and output some statistics:

1. In `Program.Helpers.cs`, import the namespaces for working with ADO.NET and common collections, as shown in the following code:

```
using Microsoft.Data.SqlClient; // To use
SqlConnection. using System.Collections;
// To use IDictionary.
```

2. In `Program.Helpers.cs`, in the partial `Program` class, add a method to output statistics about the current connection, with an array of string values to control which of the dozen or more statistics we want to output, as shown in the following code:

```
private static void
OutputStatistics(SqlConnection connection)
{ // Remove all the string values to see
  all the statistics. string[] includeKeys =
  { "BytesSent", "BytesReceived",
    "ConnectionTime", "SelectRows" };
  IDictionary statistics =
  connection.RetrieveStatistics(); foreach
  (object? key in statistics.Keys) { if (!
  includeKeys.Any() ||
  includeKeys.Contains(key)) { if
  (int.TryParse(statistics[key]?.ToString(),
  out int value)) {
  WriteLineInColor($"{key}: {value:N0}",
  ConsoleColor.Cyan); } } } }
```

3. In `Program.cs`, after writing the SQL Server version to the console, add a statement to enable statistics for the connection, as shown highlighted in the following code:

```
WriteLine($"SQL Server version:
{connection.ServerVersion}");
connection.StatisticsEnabled = true;
```

4. In `Program.cs`, before closing the connection, add a statement to output statistics for the connection, as shown highlighted in the following code:

```
OutputStatistics(connection);  
connection.Close();
```

5. Run the console app and note the statistics, as shown in the following partial output:

```
BytesReceived: 3,888 BytesSent: 336  
SelectRows: 77 ExecutionTime: 25
```

Working with ADO.NET asynchronously

You can improve the responsiveness of data access code by making it asynchronous.

Let's see how to change the statements to work asynchronously:

1. In `Program.cs`, change the statement to open the connection to make it asynchronous, as shown highlighted in the following code:

```
awaitAsync connection.Open();
```

2. In `Program.cs`, change the statement to execute the command to make it asynchronous, as shown highlighted in the following code:

```
using SqlDataReader r = awaitAsync  
cmd.ExecuteReader();
```

3. In `Program.cs`, change the statements to read the next row and get the field values to make them asynchronous, as shown highlighted in the following code:

```
while (awaitAsync r.Read()) { WriteLine(" |  
{0,5} | {1,-35} | {2,8:C} |",  
awaitFieldValueAsync<int>  
r.Get("ProductId"),  
awaitFieldValueAsync<string>  
r.Get("ProductName"),  
awaitFieldValueAsync<decimal>  
r.Get("UnitPrice")); }
```

4. In `Program.cs`, change the statements to close the data reader and connection to make them asynchronous, as shown highlighted in the following code:

```
awaitAsync r.Close(); awaitAsync  
connection.Close();
```

5. Run the console app and confirm that it has the same results as before. However, it would run better in a multithreaded system as it would, for example, not block the user interface in a GUI app, and not block I/O threads in a website.

Executing stored procedures using ADO.NET

If you need to execute the same query or another SQL statement multiple times, it is best to create a **stored procedure**, often with parameters, so that it can be precompiled and optimized. Stored procedure parameters have a direction to indicate if they are inputs, outputs, or return values.

Let's see an example that uses all three types of parameter direction. First, we will create the stored procedure in the database:

1. In your preferred database tool, connect to the `Northwind` database.
2. Add a new stored procedure:
 - If you are using **SQL Server Management Studio**, then in **Object Explorer**, navigate to **Databases | Northwind | Programmability**, right-click **Stored Procedures**, and select **New | Stored Procedure**.
 - If you are using Visual Studio, then in **Server Explorer**, right-click **Stored Procedures** and select **Add New Stored Procedure**.
 - If you are using VS Code, then in **SQL Server**, right-click your connection profile and select **New Query**.
 - If you are using Rider, then in the **Database** toolbar, click the **Jump to Query Console...** button, and then remove any existing statements. As well as the following SQL statements, start with a command to set the active database to Northwind: `USE Northwind GO`. This should prevent Rider from creating the stored procedure in the `master` database!
7. Modify the SQL statements to define a stored procedure named `GetExpensiveProducts` with two parameters, an input parameter for the minimum unit price and an output parameter for the row count of matching products, as shown in the following code:

```
CREATE PROCEDURE [dbo].  
[GetExpensiveProducts] @price money,  
@count int OUT AS PRINT 'Getting expensive  
products: ' + TRIM(CAST(@price AS  
NVARCHAR(10))) SELECT @count = COUNT(*)  
FROM Products WHERE UnitPrice >= @price  
SELECT * FROM Products WHERE UnitPrice >=
```

```
@price RETURN 0
```

The stored procedure uses two `SELECT` statements. The first sets the `@count` output parameter to a count of the matching product rows. The second returns the matching product rows. It also calls the `PRINT` command, which will raise the `InfoMessage` event.

1. Right-click in the SQL statements and select **Execute** or **Execute Query**.
2. Right-click **Stored Procedures** and select **Refresh**. In JetBrains Rider, it is named **routines**.
3. Expand **GetExpensiveProducts** and note the `@price` money input, `@count` int input/output, and return value parameters, as shown in **SQL Server Management Studio** in *Figure 7.4*:

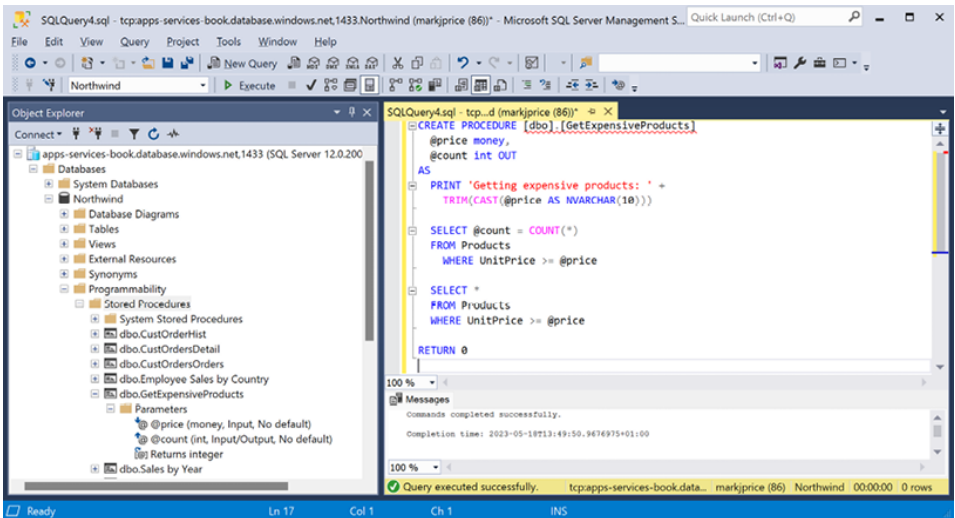


Figure 7.4: Parameters of the GetExpensiveProducts stored procedure

1. Close the SQL query without saving changes.

2. In Program.cs, add statements to allow the user to choose between running the text command and the stored procedure. Add statements defining the stored procedure and its parameters, and then execute the command, as shown highlighted in the following code:

```
using SqlCommand cmd =
connection.CreateCommand();
WriteLine("Execute command using:");
WriteLine(" 1 - Text"); WriteLine(" 2 -
Stored Procedure"); WriteLine();
Write("Press a key: "); key =
ReadKey().Key; WriteLine(); WriteLine();
SqlParameter p1, p2 = new(), p3 = new();
if (key is ConsoleKey.D1 or
ConsoleKey.NumPad1) { cmd.CommandType =
CommandType.Text; cmd.CommandText =
"SELECT ProductId, ProductName, UnitPrice
FROM Products" + " WHERE UnitPrice >=
@minimumPrice";
cmd.Parameters.AddWithValue("minimumPrice",
price); } else if (key is ConsoleKey.D2 or
ConsoleKey.NumPad2) { cmd.CommandType =
CommandType.StoredProcedure;
cmd.CommandText = "GetExpensiveProducts";
p1 = new() { ParameterName = "price",
SqlDbType = SqlDbType.Money, SqlValue =
price }; p2 = new() { Direction =
ParameterDirection.Output, ParameterName =
"count", SqlDbType = SqlDbType.Int }; p3 =
new() { Direction=
ParameterDirection.ReturnValue,
ParameterName = "rv", SqlDbType =
SqlDbType.Int };
cmd.Parameters.AddRange(new[] { p1, p2, p3
}); } using SqlDataReader r = await
```

```
cmd.ExecuteReaderAsync();
```

3. After the statement that closes the data reader, add statements to check if the user chose to use the stored procedure and if so, to output the output parameter and the return value, as shown highlighted in the following code:

```
await r.CloseAsync(); if (key is  
ConsoleKey.D2 or ConsoleKey.NumPad2) {  
WriteLine($"Output count: {p2.Value}");  
WriteLine($"Return value: {p3.Value}"); }  
await connection.CloseAsync();
```

If a stored procedure returns result sets as well as parameters, then the data reader for the result sets must be closed before the parameters can be read.

1. Run the console app and note the results if the price entered is 60, and note the `InfoMessage` event handler writes a message in dark blue, as shown in the following output:

```
Enter a unit price: 60 Execute command  
using: 1 - Text 2 - Stored Procedure Press  
a key: 2 Info: Getting expensive products:  
60.00.
```

```
-----  
| Id | Name | Price |
```

```
-----  
| 9 | Mishi Kobe Niku | $97.00 | | 18 |  
Carnarvon Tigers | $62.50 | | 20 | Sir  
Rodney's Marmalade | $81.00 | | 29 |
```

```
Thüringer Rostbratwurst | $123.79 | | 38 |  
Côte de Blaye | $263.50 |
```

```
Output count: 5 Return value: 0 State  
change from Open to Closed.
```

Managing transactions with ADO.NET

In ADO.NET, a transaction ensures that a set of database operations executes as an atomic unit. This means either all operations succeed and are committed, or if an error occurs, the entire transaction is rolled back, leaving the database in its original state.

Key components of transactions in ADO.NET

Transactions in ADO.NET are managed using the `BeginTransaction`, `Commit`, and `Rollback` methods provided by the `SqlConnection` class.

To initiate a transaction, you use the `BeginTransaction` method of the `SqlConnection` object. This returns a `SqlTransaction` object, which is then used to commit or roll back changes.

If all operations within the transaction complete successfully, you call the `Commit` method to make the changes permanent.

If an error occurs at any point, you should call `Rollback` to undo all the changes made within the transaction.

For example, as shown in the following code:

```
using System.Data; using  
System.Data.SqlClient; string connectionString  
= "your_connection_string_here"; using  
SqlConnection conn = new(connectionString);  
conn.Open(); // Start a transaction on the
```

```

open database connection. SqlTransaction
transaction = conn.BeginTransaction(); try {
// Create a command and associate it with the
transaction. using (SqlCommand cmd1 =
new("INSERT INTO Products (ProductName,
UnitPrice) VALUES ('Mint Tea', 29.99)", conn,
transaction)) { cmd1.ExecuteNonQuery(); }
using (SqlCommand cmd2 = new("INSERT INTO
Orders (ProductId, OrderDate) VALUES
(SCOPE_IDENTITY(), GETDATE())", conn,
transaction)) { cmd2.ExecuteNonQuery(); } //
If everything is successful, commit the
transaction. transaction.Commit();
WriteLine("Transaction committed
successfully."); } catch (Exception ex) { //
If an error occurs, roll back the transaction.
transaction.Rollback(); WriteLine("Transaction
rolled back due to an error: " + ex.Message);
}

```

Note the following about the preceding code:

1. The `SqlConnection` object named `conn` is created and opened.
2. `conn.BeginTransaction()` is called to start a transaction. This returns a `SqlTransaction` object, which is then passed to `SqlCommand` to associate the commands with the transaction. A `using` operator is omitted here on purpose to demonstrate the need to call a rollback in a `catch` block.
3. The first command inserts a new product into the `Products` table.
4. The second command inserts an order for the newly added product. `SCOPE_IDENTITY()` is used to retrieve the last inserted `ProductId` value.
5. If everything succeeds, `transaction.Commit()` is called.
6. If any error occurs, the `catch` block rolls back the transaction using

`transaction.Rollback()`. This explicit call is necessary because the transaction is not wrapped in a `using` block. If it were, the rollback would happen implicitly when the transaction scope ends.

Handling transactions with using blocks

The above example requires explicit calls to `Rollback` because there is no `using` statement. `SqlTransaction` implements the `IDisposable` interface and calls `Rollback` in THE `Dispose` method if the transaction has not already been committed. So, you can simplify transaction handling using the `using` statement for automatic disposal, as shown in the following code:

```
using SqlConnection conn =
new(connectionString); conn.Open(); using
SqlTransaction transaction =
conn.BeginTransaction(); using (SqlCommand cmd
= new("SQL QUERY HERE", conn, transaction)) {
cmd.ExecuteNonQuery(); } transaction.Commit();
```

This ensures that the transaction is properly disposed of, even in case of exceptions.

Isolation levels in transactions

When calling `BeginTransaction`, you can specify an isolation level to control how changes are visible to other transactions, as shown in *Table 7.8* and in the following code:

```
SqlTransaction transaction =
conn.BeginTransaction(IsolationLevel.Serializable);
```

Isolation level	Repeatable reads	Serializable
Read uncommitted		
Read committed		
Repeatable reads		
Serializable		

Read Committed (default)			
Repeatable Reads			
Serializable			
Snapshot			

Table 7.8: Common isolation levels

When to use transactions?

Use transactions in ADO.NET when:

- Performing multiple related database operations that must succeed or fail as a unit.
- Ensuring data integrity in banking, order processing, or financial transactions.
- Preventing partial updates due to network failures or application crashes.

Transactions in ADO.NET provide a reliable way to ensure data consistency and integrity. By using `BeginTransaction()`, `Commit()`, and `Rollback()`, you can manage changes to your database in a controlled manner.

Now let's see how we can output streams with a data reader instead of writing to the console.

Outputting streams with a data reader

In a real app or service, we would likely not output to the console. More likely, as we read each row with a data reader, we might output to a stream that writes HTML tags inside a web page, or text formats like XML and JSON for returning data from a service.

Let's add the ability to generate a JSON file:

1. In `Program.cs`, import the namespace for working efficiently with JSON and to statically import the `Environment` and `Path` classes, as shown in the following code:


```
using System.Text.Json; // To use
Utf8JsonWriter, JsonSerializer. using
static System.Environment; using static
System.IO.Path;
```

2. In `Program.cs`, before the `while` statement that processes the data reader, add statements to define a file path for a JSON file, create a file stream, and start a JSON array. Then, in the `while` block, write a JSON object that represents each product row, and end the array and close the stream, as shown highlighted in the following code:

```
// Define a file path to write to. string
jsonPath = Combine(CurrentDirectory,
"products.json"); await using (FileStream
jsonStream = File.Create(jsonPath)) {
Utf8JsonWriter jsonWriter =
new(jsonStream);
jsonWriter.WriteStartArray(); while (await
r.ReadAsync()) { WriteLine("| {0,5} |
{1,-35} | {2,10:C} |", await
r.GetFieldValueAsync<int>("ProductId"),
await
r.GetFieldValueAsync<string>("ProductName"),
await
r.GetFieldValueAsync<decimal>("UnitPrice"));
jsonWriter.WriteStartObject();
jsonWriter.WriteNumber("productId", await
r.GetFieldValueAsync<int>("ProductId"));
jsonWriter.WriteString("productName",
await
r.GetFieldValueAsync<string>("ProductName"));
jsonWriter.WriteNumber("unitPrice", await
r.GetFieldValueAsync<decimal>("UnitPrice"));
jsonWriter.WriteEndObject(); }
```

```
jsonWriter.WriteEndArray();  
jsonWriter.Flush(); jsonStream.Close(); }  
WriteLineInColor($"Written to:  
{jsonPath}", ConsoleColor.DarkGreen);
```

3. Run the console app, enter a price of 60, and note the path to the JSON file, as shown in the following output:

```
Written to: C:\apps-services-  
net10\Northwind.Blazor  
\Northwind.Console.SqlClient\bin\Debug  
\net10.0\products.json
```

4. Open the `products.json` file and note that the JSON is written with no whitespace, so it all appears on one line, as shown in the following file:

```
[{"productId":9,"productName":"Mishi Kobe  
Niku","unitPrice":97.0000},  
{"productId":18,"productName":"Carnarvon  
Tigers","unitPrice":62.5000},  
{"productId":20,"productName":"Sir Rodney  
\u0027s Marmalade","unitPrice":81.0000},  
{"productId":29,"productName":"Th  
\u00FCringer  
Rostbratwurst","unitPrice":123.7900},  
{"productId":38,"productName":"C\u00F4te  
de Blaye","unitPrice":263.5000}]
```

5. If you are using Visual Studio, then you can right-click and select **Format Document**, and note that it is now easier to read, as shown in *Figure 7.5*:

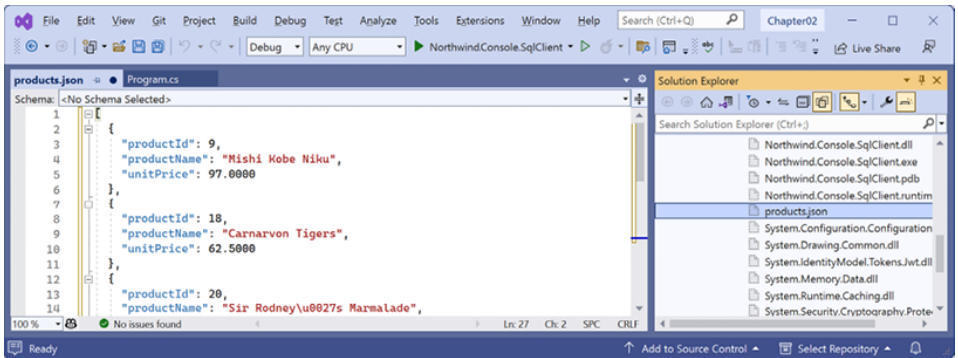


Figure 7.5: The products.json file generated from a data reader

Generating objects with a data reader

For maximum flexibility, we likely want to convert the rows in a data reader into object instances stored in an array or collection. After that, we could serialize the object graph however we want. ADO.NET does not have a built-in ability to map a data reader row to an object, so we will have to do it manually.

Let's see an example:

1. Add a new class file named `Product.cs`, and modify its contents to define a class to represent just the three columns we want from each row in the `Products` table, as shown in the following code:

```
namespace Northwind.Models; public class
Product { public int ProductId { get; set;
} public string? ProductName { get; set; }
public decimal? UnitPrice { get; set; } }
```

Good practice: In this task, we will use this type only for read-only instances, so we could have used `init` instead of `set` for an immutable record. But later

we will need to change property values after the object is created, so we must define a class with set-able properties instead.

1. At the top of `Program.cs`, import the `Northwind.Models` namespace so we can use `Product`.
2. In `Program.cs`, before creating the file stream, instantiate a list of products with an initial storage for 77 items (but this is not a limit) because when first created the Northwind database has 77 products, as shown highlighted in the following code:

```
List<Product> products = new(capacity:
77); await using (FileStream jsonStream =
File.Create(jsonPath))
```

3. In the `while` block, add statements to instantiate the `Product` type per row in the data reader and add it to the list, as shown highlighted in the following code:

```
while (await r.ReadAsync()) { Product
product = new() { ProductId = await
r.GetFieldValueAsync<int>("ProductId"),
ProductName = await
r.GetFieldValueAsync<string>("ProductName"),
UnitPrice = await
r.GetFieldValueAsync<decimal>("UnitPrice")
}; products.Add(product); ... }
```

4. Before closing the data reader, add a statement to use the static `Serialize` method of the `JsonSerializer` class to write the list of products to the console, as shown highlighted in the following code:

```
WriteLineInColor(JsonSerializer.Serialize(product),
ConsoleColor.Magenta); await
r.CloseAsync();
```

5. Run the console app, enter a price of 60, and note the JSON generated from the list of products, as shown in the following output:

```
Written to: C:\apps-services-
net10\Northwind.Blazor
\Northwind.Console.SqlClient\bin\Debug
\net10.0\products.json
[{"ProductId":9,"ProductName":"Mishi Kobe
Niku","UnitPrice":97.0000},
{"ProductId":18,"ProductName":"Carnarvon
Tigers","UnitPrice":62.5000},
{"ProductId":20,"ProductName":"Sir Rodney
\u0027s Marmalade","UnitPrice":81.0000},
{"ProductId":29,"ProductName":"Th
\u00FCringer
Rostbratwurst","UnitPrice":123.7900},
{"ProductId":38,"ProductName":"C\u00F4te
de Blaye","UnitPrice":263.5000}]
```

Instead of manually instantiating objects, to simplify even more, we can use a simple **object-relational mapper (ORM)** like Dapper.

Managing data with Dapper

Dapper uses ADO.NET underneath when working with SQL Server. Because it is a higher-level technology, it is not as efficient as using ADO.NET directly, but it can be easier. Dapper is an alternative ORM to EF Core. It is more efficient because it extends the low-level ADO.NET `IDbConnection`

interface with very basic functionality without trying to be all things to all people.

Dapper connection extension methods

Dapper adds three extension methods to any class that implements `IDbConnection` (like `SqlConnection`). They are `Query<T>`, `Query`, and `Execute`, along with their asynchronous equivalents. Dapper will automatically open and close the associated connection as needed.

The `Query<T>` extension method is the most used because it runs any specified SQL command and then returns the results as an `IEnumerable<T>` (a sequence of objects). It is designed to run commands that retrieve data like `SELECT`. It has several parameters, as shown in *Table 7.9*:

Description		
This is the only mandatory parameter. It is either the text of a SQL command or the name of a stored procedure.		
An object or object for passing parameters used in the query. This can be an anonymous type.		
To manage transactions. <code>transaction = null</code> .		
By default, it will buffer the entire reader on return. With large datasets, you can minimize memory and only load objects as needed by setting <code>buffered to false</code> .		
To change the default command timeout.		
To switch to a stored procedure instead of the default of text.		

Table 7.9: Dapper’s `Query<T>` extension method parameters

The `Query` extension method is a loosely-typed equivalent so it is less frequently used.

The `Execute` extension method runs any specified SQL command and then returns the number of rows affected as an `int`. It is designed to run commands like `INSERT`, `UPDATE`, and `DELETE`. It has the same parameters as the

Query<T> extension method.

Querying using Dapper

Let's see a simple example that queries the `Suppliers` table instead of the `Products` table:

1. In the `Northwind.Console.SqlClient` project, add a package reference for `Dapper`, as shown highlighted in the following markup:

```
<ItemGroup> <PackageReference  
Include="Microsoft.Data.SqlClient" />  
<PackageReference Include="Dapper" /> </  
ItemGroup>
```

2. Build the project to restore packages.
3. Add a new class file named `Supplier.cs`, and modify its contents to define a class to represent four columns from each row in the `Suppliers` table, as shown in the following code:

```
namespace Northwind.Models; public class  
Supplier { public int SupplierId { get;  
set; } public string? CompanyName { get;  
set; } public string? City { get; set; }  
public string? Country { get; set; } }
```

4. At the bottom of `Program.cs`, add statements to retrieve `Supplier` entities in `Germany`, enumerate the collection outputting basic information about each one, and then serialize the collection as JSON to the console, as shown in the following code:

```
WriteLineInColor("Using Dapper",
```

```

ConsoleColor.DarkGreen);
connection.ResetStatistics(); // So we can
compare using Dapper.
IEnumerable<Supplier> suppliers =
connection.Query<Supplier>( sql: "SELECT *
FROM Suppliers WHERE Country=@Country",
param: new { Country = "Germany" });
foreach (Supplier s in suppliers) {
WriteLine("{0}: {1}, {2}, {3}",
s.SupplierId, s.CompanyName, s.City,
s.Country); }
WriteLineInColor(JsonSerializer.Serialize(suppliers),
ConsoleColor.Green);
OutputStatistics(connection);

```

5. Run the console app, and in the section where we used Dapper, note the same connection was used, so its events were raised while the Dapper query was executed, the enumerated collection output, and then JSON generated from the list of suppliers, as shown in the following output:

```

Using Dapper 11: Heli Süßwaren GmbH & Co.
KG, Berlin, Germany 12: Plutzer
Lebensmittelgroßmärkte AG, Frankfurt,
Germany 13: Nord-Ost-Fisch
Handelsgesellschaft mbH, Cuxhaven, Germany
[{"SupplierId":11, "CompanyName":"Heli S
\u00FC\u00DFwaren GmbH \u0026 Co. KG",
"City":"Berlin","Country":"Germany"},
{"SupplierId":12, "CompanyName":"Plutzer
Lebensmittelgro\u00DFm\u00E4rkte AG",
"City":"Frankfurt","Country":"Germany"},
{"SupplierId":13, "CompanyName":"Nord-Ost-
Fisch Handelsgesellschaft mbH",
"City":"Cuxhaven","Country":"Germany"}]

```



```
BytesReceived: 1,430 BytesSent: 240  
SelectRows: 3 ExecutionTime: 5
```

6. At the bottom of `Program.cs`, add statements to run the `GetExpensiveProducts` stored procedure, passing a price parameter value of 100, enumerate the collection and outputting of basic information about each item, and then serialize the collection as JSON to the console, as shown in the following code:

```
IEnumerable<Product> productsFromDapper =  
connection.Query<Product>(sql:  
    "GetExpensiveProducts", param: new { price  
    = 100M, count = 0 }, commandType:  
    CommandType.StoredProcedure); foreach  
    (Product p in productsFromDapper) {  
    WriteLine("{0}: {1}, {2}", p.ProductId,  
    p.ProductName, p.UnitPrice); }  
WriteLineInColor(JsonSerializer.Serialize(productsFromDapper),  
    ConsoleColor.Green);
```

Warning! With Dapper, you must pass a param object with all parameters, even if they are only used as output parameters. For example, we must define `count`, or an exception will be thrown. You must also remember to explicitly set the command type to stored procedure!

1. Run the console app, and in the section where we used Dapper to run the stored procedure to get the products that cost more than 100, note the same connection was used so its events were raised while the Dapper query was executed, the enumerated collection output, and then JSON

generated from the list of products, as shown in the following output:

```
Info: Getting expensive products: 100.00.  
29: Thüringer Rostbratwurst, 123.7900 38:  
Côte de Blaye, 263.5000  
[{"ProductId":29,"ProductName":"Th  
\u00FCringer  
Rostbratwurst","UnitPrice":123.7900},  
{"ProductId":38,"ProductName":"C\u00F4te  
de Blaye","UnitPrice":263.5000}]
```

You can learn more about Dapper at the following link: <https://github.com/DapperLib/Dapper/blob/main/Readme.md>.

Practicing and exploring

Test your knowledge and understanding by answering some questions, getting some hands-on practice, and exploring this chapter's topics with deeper research.

Exercise 7.1 – Online material

Online material can be extra content written by me for this book, or it can be references to content created by Microsoft or third parties.

Alternatives for storing secrets

Secrets like passwords and other values used in database connection strings, or values like keys to access a service, are often stored in environment variables. Other places for storing these values include App Secrets. You can learn more about them in the article *Safe storage of app secrets in development in ASP.NET*

Core, found at the following link: <https://learn.microsoft.com/en-us/aspnet/core/security/app-secrets>.

For related guidance about handling connection strings, you can read the following link: <https://learn.microsoft.com/en-us/ef/core/miscellaneous/connection-strings>.

To use Azure Key Vault to store your secrets: <https://learn.microsoft.com/en-us/azure/key-vault/secrets/quick-create-cli>.

App secret recommendations: <https://learn.microsoft.com/en-us/training/modules/aspnet-configurationbuilder/?source=recommendations>.

ASP.NET Core secret manager: <https://learn.microsoft.com/en-us/aspnet/core/security/app-secrets/#secret-manager>.

Exercise 7.2 – Test your knowledge

Answer the following questions:

1. Which NuGet package should you reference in a .NET project to get the best performance when working with data in SQL Server?
2. What is the safest way to define a database connection string?
3. What must T-SQL parameters and variables be prefixed with?
4. What must you do before reading an output parameter of a command executed using `ExecuteReader`?
5. What type does Dapper add its extension methods to?
6. What are the two most commonly used extension methods provided by Dapper?

Appendix A, Answers to the Test Your Knowledge Questions, is available to download from the following link: <https://github.com/>

[markjprice/apps-services-net10/
blob/main/docs/B31467_Appendix
%20A.pdf](https://github.com/markjprice/apps-services-net10/blob/main/docs/B31467_Appendix%20A.pdf).

Exercise 7.3 – Explore topics

Use the links on the following page to learn more details about the topics covered in this chapter: <https://github.com/markjprice/apps-services-net10/blob/main/docs/book-links.md#chapter-7---managing-relational-data-using-sql>.

Summary

In this chapter, you learned:

- How to connect to an existing SQL Server database.
- How to execute a simple query and process the results using fast and low-level ADO.NET.
- How to execute a simple query and process the results using Dapper.

In the next chapter, you will learn how to use the more powerful and complex ORM from Microsoft named EF Core.

Get This Book **UNLOCK NOW** and Exclusive Extras

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.



Note: *Keep your invoice handy. Purchases made directly from Packt don't require one.*

8

Building Entity Models Using EF Core

This chapter is about managing relational data stored in SQL Server by using the higher-level object-to-data store mapping technology named **Entity Framework Core (EF Core)**. You will review how to define entity models, and how to query a model. Next, you will learn how EF Core tracks changes in its data context so that it can later add, update and delete data in the underlying database. Then, you will learn how to store entity models that use inheritance hierarchies using three different mapping strategies.

This chapter will cover the following topics:

- Managing data with EF Core
- Controlling the tracking of entities
- Mapping inheritance hierarchies with EF Core

Managing data with EF Core

EF Core is an **object-relational mapper (ORM)** that uses ADO.NET underneath when working with SQL Server. Because it is a higher-level technology, it is not as efficient as using ADO.NET directly, but it can be easier for developers to work with because they can treat the data as objects instead of

rows in multiple tables. This should feel more natural for an object-oriented developer.

EF Core 10 only targets .NET 10. EF Core 9 targeted .NET 8, so it could be used with both the **Long Term Support (LTS)** release of .NET 8 and the **Standard Term Support (STS)** release of .NET 9, as shown in *Figure 8.1*:

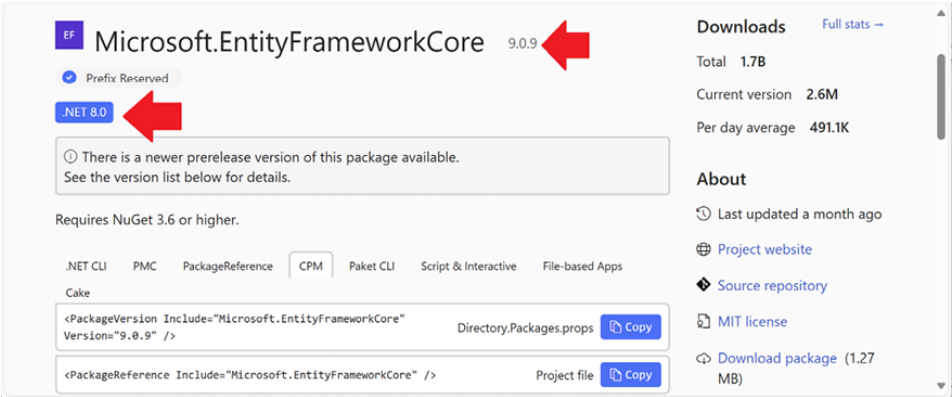


Figure 8.1: EF Core 9 targets .NET 8 or later

When EF Core 11 is released in November 2026, we can expect it to target .NET 10 or later, so you can upgrade EF Core while still getting long-term support for the .NET 10 platform. The EF Core team is responsible for making sure that you will be able to swap 10 for 11 in the version number of their packages and your code should still work. They are usually very good at that, and they will document any needed changes to your code in the official release notes for EF Core 11.

Working with EF Core

As well as traditional RDBMSs like SQL Server, EF Core supports modern cloud-based, nonrelational, schema-less data stores, such as Azure Cosmos DB and MongoDB, sometimes with third-party providers.

There are two approaches to working with EF Core:

- **Database First:** A database already exists, so you build a model that

matches its structure and features.

- **Code First:** No database exists, so you build a model and then use EF Core to create a database that matches its structure and features.

We will use EF Core with an existing database because that is the most common scenario. You will also see an example of Code First later in this chapter, that will create its database at runtime.

Scaffolding models using an existing database

Scaffolding is the process of using a tool to create classes that represent the model of an existing database using reverse engineering. A good scaffolding tool allows you to extend the automatically generated classes and then regenerate those classes without losing your extended classes. You used the `dotnet-ef` tool to build the reusable Northwind entity model in [Chapter 1, *Introducing Apps and Services with .NET*](#).

If you know that you will never regenerate the classes using the tool, then feel free to change the code for the automatically generated classes as much as you want. The code generated by the tool is just the best approximation.

Good practice: Do not be afraid to overrule a tool when you know better. For example, when using SQLite for the Northwind database, date/time columns are mapped to `string` properties, and money columns are mapped to `double` properties. In the Northwind database, these would be better mapped to `DateTime` and `decimal` respectively, but in another database, they might need more flexibility. As another example, in the Northwind database, a `CustomerId` should always be five uppercase alphabetic characters. The

tool cannot infer this automatically, so you could add a regular expression to validate it. Keep in mind that tool behavior is one of the more volatile components of .NET, so these examples may not be valid by the time you read this book.

Defining EF Core models

EF Core uses a combination of **conventions**, **annotation attributes**, and **Fluent API** statements to build an **entity model** at runtime so that any actions performed on the classes can later be automatically translated into actions performed on the actual database. An **entity class** represents the structure of a table, and an instance of the class represents a row in that table.

First, we will review the three ways to define a model, with code examples, and then we will create some classes that implement these techniques.

Using EF Core conventions to define the model

The code we will write will use the following conventions:

- The name of a table is assumed to match the name of a `DbSet<T>` property in the `DbContext` class, for example, `Products`.
- The names of the columns are assumed to match the names of properties in the entity model class, for example, `ProductId`.
- The `string` .NET type is assumed to be an `nvarchar(max)` type in the database. You should use the `[StringLength]` annotation (or the equivalent fluent API call) to limit the number of characters in the `string` property because `nvarchar(max)` can store up to 2 GB of characters.
- The `int` .NET type is assumed to be an `int` type in the database.
- The primary key is assumed to be a property that is named `Id` or `ID`. Or, when the entity model class is named `Product`, then the property

can be named `ProductId` or `ProductID`. If this property is of an integer type or the `Guid` type, then it is also assumed to be an `IDENTITY` column (a column type that automatically assigns a value when inserting).

Good practice: There are many other conventions that you should know, and you can even define your own, but that is beyond the scope of this book. You can read about them at the following link: <https://learn.microsoft.com/en-us/ef/core/modeling/>.

Using EF Core annotation attributes to define the model

Conventions often aren't enough to completely map the classes to the database objects. For example, some databases like SQLite use dynamic column types, so the tool has to guess what property types its columns should map to, based on the current data values in that column.

A simple way of adding more smarts to your model is to apply annotation attributes, as shown in *Table 8.1*:

Description		
<code>IsRequired()</code>		Ensures the value is not null
<code>HasMaxLength(50)</code>		Ensures the value is (up to) 50 characters in length
<code>Regex(@"[a-z0-9]+")</code>		Ensures the value matches the specified regular expression
<code>HasColumnName("rice")</code>		Specifies the column name used in the table

Table 8.1: Common EF Core annotation attributes

For example, in the database, the maximum length of a product name is 40, and

the value cannot be null, as shown highlighted in the following partial DDL code, which defines how to create a table named `Products`:

```
CREATE TABLE Products ( ProductId INTEGER
PRIMARY KEY, ProductName NVARCHAR (40) NOT
NULL, SupplierId "INT", ... );
```

In a `Product` class, we could apply attributes to specify this, as shown in the following code:

```
[Required] [StringLength(40)] public string
ProductName { get; set; }
```

Good practice: If you have nullability checks enabled, then you do not need to decorate a non-nullable reference type with the `[Required]` attribute as shown above. This is because the C# nullability will flow to the EF Core model. A `string` property will be required; a `string?` property will be optional, in other words, nullable. You can read more about this at the following link: <https://learn.microsoft.com/en-us/ef/core/modeling/entity-properties?tabs=data-annotations%2Cwith-nrt#required-and-optional-properties>.

When there isn't an obvious map between .NET types and database types, an attribute can be used.

For example, in the database, the column type of `UnitPrice` for the `Products` table is `money`. .NET does not have a `money` type, so it should

use `decimal` instead, as shown in the following code:

```
[Column(TypeName = "money")] public decimal?
UnitPrice { get; set; }
```

Another example is for the `Categories` table, as shown highlighted in the following DDL code:

```
CREATE TABLE Categories ( CategoryId INTEGER
PRIMARY KEY, CategoryName NVARCHAR (15) NOT
NULL, Description "NTEXT", Picture "IMAGE" );
```

The `Description` column can be longer than the maximum 8,000 characters that can be stored in an `nvarchar` variable, so it needs to map to `ntext` instead, as shown in the following code:

```
[Column(TypeName = "ntext")] public string?
Description { get; set; }
```

Using the EF Core Fluent API to define the model

The last way that the model can be defined is by using the Fluent API. This API can be used instead of attributes, as well as being used in addition to them. One reason you might need to do this is if the entity models need to be defined in a .NET Standard 2.0 class library so they can be used on legacy platforms. In this case, the class library should not include references to data annotation libraries. Another reason is that your team might have a policy to separate raw data models from validation rules.

For example, to define the `ProductName` property, instead of decorating the

property with two attributes, an equivalent Fluent API statement could be written in the `OnModelCreating` method of the database context class, as shown in the following code:

```
modelBuilder.Entity<Product>()  
    .Property(product => product.ProductName)  
    .IsRequired() // only needed if you have  
    disabled nullability checks .HasMaxLength(40);
```

This keeps the entity model class simpler. You will see an example of this in the coding task below.

Understanding data seeding

Another benefit of the Fluent API is to provide initial data to populate a database. EF Core automatically works out what insert, update, or delete operations must be executed.

For example, if we wanted to make sure that a new database has at least one row in the `Product` table, then we would call the `HasData` method, as shown in the following code:

```
modelBuilder.Entity<Product>() .HasData(new  
    Product { ProductId = 1, ProductName = "Chai",  
    UnitPrice = 8.99M });
```

Calls to `HasData` take effect either during a data migration executed by the `dotnet ef database update` command or when you call the `Database.EnsureCreated` method.

Our model will map to an existing database that is already populated with data, so we will not need to use this technique in our code.

Querying the Northwind database model

A `NorthwindContext` class is used to represent the database. To use EF Core, the class must inherit from `DbContext`. This class understands how to communicate with databases and dynamically generate SQL statements to query and manipulate data.

Your `DbContext`-derived class should have an overridden method named `OnConfiguring`, which will set the database connection string.

Inside your `DbContext`-derived class, you must define at least one property of the `DbSet<T>` type. These properties represent the tables. To tell EF Core what columns each table has, the `DbSet<T>` properties use generics to specify a class that represents a row in the table. That entity model class has properties that represent its columns.

The `DbContext`-derived class can optionally have an overridden method named `OnModelCreating`. This is where you can write Fluent API statements as an alternative to decorating your entity classes with attributes. This can enhance the clarity and maintainability of the model configuration because all of it could be in one place instead of scattered throughout your code base. You can also mix and match both techniques, but then you'd lose this primary benefit.

Let's use the model for the Northwind database that you created in [Chapter 1, *Introducing Apps and Services with .NET*](#), in a console app:

1. Use your preferred code editor to create a console app project, as defined in the following list:
 - Project template: **Console App** / `console`
 - Solution file and folder: `ModernApps`
 - Project file and folder:
`Northwind.Console.EFCore`
 - **Do not use top-level statements**: Cleared
 - **Enable native AOT publish**: Cleared

7. In the `Northwind.Console.EFCore` project, add a reference to the `Northwind` database context project, and globally and statically import the `System.Console` class, as shown highlighted in the following markup:

```
<ItemGroup> <ProjectReference Include="..\
  \Northwind.DataContext
  \Northwind.DataContext.csproj" /> </
ItemGroup> <ItemGroup Label="To simplify
use of WriteLine."> <Using
Include="System.Console" Static="true" />
</ItemGroup>
```

8. Build the project to build and restore referenced projects.
9. In `Northwind.EntityModels`, in `Category.cs`, note that it represents a row in the `Categories` table, as shown in the following code:

```
using System; using
System.Collections.Generic; using
System.ComponentModel.DataAnnotations;
using
System.ComponentModel.DataAnnotations.Schema;
using Microsoft.EntityFrameworkCore;
namespace Northwind.Models;
[Index("CategoryName", Name =
"CategoryName")] public partial class
Category { [Key] public int CategoryId {
get; set; } [StringLength(15)] public
string CategoryName { get; set; } = null!;
[Column(TableName = "ntext")] public
string? Description { get; set; }
[Column(TableName = "image")] public
```

```
byte[]? Picture { get; set; }  
[InverseProperty("Category")] public  
virtual ICollection<Product> Products {  
get; set; } = new List<Product>(); }
```

Note the following:

- It decorates the entity class with the `[Index]` attribute that was introduced in EF Core 5. This indicates properties that should have an index for their column in the table. In earlier versions, only the Fluent API was supported for defining indexes. Since we are working with an existing database, this attribute is not needed. But if we want to recreate a new empty Northwind database from our code, then this information will be used to create indexes in the `Categories` table.
- The table name in the database is `Categories`, but the `dotnet-ef` tool used the **Humanizer** third-party library to automatically singularize the class name to `Category`, which is a more natural name when creating a single entity.
- The entity class is declared using the `partial` keyword so that you can create a matching `partial` class for adding additional code. This allows you to rerun the tool and regenerate the entity class without losing that extra code you wrote in your `partial` class.
- The `CategoryId` property is decorated with the `[Key]` attribute to explicitly indicate that it is the primary key for this entity, although its name follows the primary key convention too.
- The `Products` property uses the `[InverseProperty]` attribute to define the foreign key relationship to the `Category` property on the `Product` entity class.

1. In `ProductsAboveAveragePrice.cs`, note that it represents a

row returned by a database view rather than a table, so it is decorated with the [Keyless] attribute.

2. In `NorthwindContext.cs`, review the class, as shown in the following edited-for-space code:

```
using System; using
System.Collections.Generic; using
Microsoft.EntityFrameworkCore; namespace
Northwind.Models; public partial class
NorthwindDb : DbContext { public
NorthwindDb() { } public
NorthwindDb(DbContextOptions<NorthwindDb>
options) : base(options) { } public
virtual DbSet<AlphabeticalListOfProduct>
AlphabeticalListOfProducts { get; set; }
public virtual DbSet<Category> Categories
{ get; set; } ... public virtual
DbSet<Territory> Territories { get; set; }
protected override void OnConfiguring(
DbContextOptionsBuilder optionsBuilder) {
... } protected override void
OnModelCreating(ModelBuilder modelBuilder)
{
modelBuilder.Entity<AlphabeticalListOfProduct>(en
=> { entity.ToView("Alphabetical list of
products"); }); ...
OnModelCreatingPartial(modelBuilder); }
partial void
OnModelCreatingPartial(ModelBuilder
modelBuilder); }
```

Note the following:

- The `NorthwindContext` data context class is `partial` to

allow you to extend it and regenerate it in the future. We used the name `NorthwindContext` because `Northwind` is used for a namespace.

- `NorthwindContext` has two constructors: a default parameter-less one and one that allows options to be passed in. This is useful in apps where you want to specify the connection string at runtime.
- The `DbSet<T>` properties that represent tables like `Categories`.
- In the `OnConfiguring` method, if options have not been specified in the constructor, then it defaults to using the connection string used during scaffolding. It has a compiler warning to remind you that you should not hardcode security information in this connection string.
- In the `OnModelCreating` method, the Fluent API is used to configure the entity classes, and then a partial method named `OnModelCreatingPartial` is invoked. This allows you to implement that partial method in your own partial `Northwind` class to add your own Fluent API configuration, which will not be lost if you regenerate the model classes.

1. In `Program.cs`, delete the existing statements. Add statements to create an instance of the `NorthwindContext` data context class and use it to query the products table for those that cost more than a given price, as shown in the following code:

The following code is long but easy to understand. This is a good opportunity to copy blocks of code from the file in the GitHub repository instead of typing it all yourself: <https://github.com/markjprice/apps-services->

```
net10/blob/main/code/  
ModernApps/  
Northwind.Console.EFCore/  
Program.cs.
```

```
using Microsoft.Data.SqlClient; // To use  
SqlConnectionStringBuilder. using  
Microsoft.EntityFrameworkCore; //  
ToQueryString, GetConnectionString using  
Northwind.EntityModels; // To use  
NorthwindContext. SqlConnectionStringBuilder  
builder = new(); builder.InitialCatalog =  
"Northwind"; builder.MultipleActiveResultSets  
= true; builder.Encrypt = true;  
builder.TrustServerCertificate = true; //  
Because we want to fail faster. Default is 15  
seconds. builder.ConnectTimeout = 3;  
WriteLine("Connect to:"); WriteLine(" 1 - SQL  
Server on local machine"); WriteLine(" 2 -  
Azure SQL Database"); WriteLine(" 3 - SQL  
Server in a container"); WriteLine();  
Write("Press a key: "); ConsoleKey key =  
ReadKey().Key; WriteLine(); WriteLine(); if  
(key is ConsoleKey.D1 or ConsoleKey.NumPad1) {  
builder.DataSource = "."; // Local SQL Server  
// @".\apps-services-book"; // Local SQL  
Server with an instance name } else if (key is  
ConsoleKey.D2 or ConsoleKey.NumPad2) {  
builder.DataSource = // Azure SQL Database  
"tcp:apps-services-  
book.database.windows.net,1433"; } else if  
(key is ConsoleKey.D3 or ConsoleKey.NumPad3) {  
builder.DataSource = "tcp:127.0.0.1,1433"; //  
SQL Server in a container } else {
```

```
WriteLine("No data source selected."); return;
} WriteLine("Authenticate using:");
WriteLine(" 1 - Windows Integrated Security");
WriteLine(" 2 - SQL Login, for example, sa");
WriteLine(); Write("Press a key: "); key =
ReadKey().Key; WriteLine(); WriteLine(); if
(key is ConsoleKey.D1 or ConsoleKey.NumPad1) {
builder.IntegratedSecurity = true; } else if
(key is ConsoleKey.D2 or ConsoleKey.NumPad2) {
Write("Enter your SQL Server user ID: ");
string? userId = ReadLine(); if
(string.IsNullOrEmpty(userId)) {
WriteLine("User ID cannot be empty or null.");
return; } builder.UserID = userId;
Write("Enter your SQL Server password: ");
string? password = ReadLine(); if
(string.IsNullOrEmpty(password)) {
WriteLine("Password cannot be empty or
null."); return; } builder.Password =
password; builder.PersistSecurityInfo = false;
} else { WriteLine("No authentication
selected."); return; }
DbContextOptionsBuilder<NorthwindContext>
options = new();
options.UseSqlServer(builder.ConnectionString);
using (NorthwindContext db =
new(options.Options)) { Write("Enter a unit
price: "); string? priceText = ReadLine(); if
(!decimal.TryParse(priceText, out decimal
price)) { WriteLine("You must enter a valid
unit price."); return; } // We have to use var
because we are projecting into an anonymous
type. var products = db.Products .Where(p =>
p.UnitPrice > price) .Select(p => new {
p.ProductId, p.ProductName, p.UnitPrice });
```

```

WriteLine("-----");
WriteLine("| {0,5} | {1,-35} | {2,8} |", "Id",
"Name", "Price");
WriteLine("-----");
foreach (var p in products) { WriteLine("|
{0,5} | {1,-35} | {2,8:C} |", p.ProductId,
p.ProductName, p.UnitPrice); }
WriteLine("-----");
WriteLine(products.ToQueryString());
WriteLine(); WriteLine($"Provider:
{db.Database.ProviderName}");
WriteLine($"Connection:
{db.Database.GetConnectionString()}"); }

```

Note the following about the preceding code:

- **Summary:** It connects to a SQL Server database (either locally, in Azure, or in a Docker container), queries products from the Northwind sample database, and prints results to the console. It demonstrates building a SQL connection string programmatically, configuring EF Core, and dynamically running a LINQ query against the database.
- **Namespaces:** `Microsoft.Data.SqlClient`: Used to construct and manage SQL Server connection strings safely. `Microsoft.EntityFrameworkCore`: Provides EF Core APIs for building `DbContext` options and running LINQ queries. `Northwind.EntityModels`: This is a project-specific namespace that defines the `NorthwindContext` and entity classes like `Product`.
- **Connection string:** `SqlConnectionStringBuilder` creates a SQL connection string with some base options:
`InitialCatalog`: The database name (Northwind).
`MultipleActiveResultSets (MARS)`: Allows multiple

queries to be active on one connection. `Encrypt = true`: Encrypts the connection to SQL Server.

`TrustServerCertificate = true`: Accepts self-signed certificates (useful for local dev or containers).

- **Data source:** The user is asked to select where SQL Server is running, and depending on the key pressed, `builder.DataSource` is set to the appropriate value. If the user doesn't pick one, the app exits.
- **Authentication:** There are two authentication paths, Windows and SQL Server, which require a username and password. The `PersistSecurityInfo` property prevents the password from being exposed later.
- **Options:** The code creates a set of options telling EF Core to use SQL Server with the built connection string.
- **Running the query:** The database context is instantiated inside a `using` block to ensure the database connection is closed automatically. The program asks for a minimum unit price and validates it. Then it constructs a LINQ query. This translates into SQL (EF Core will generate the SQL dynamically). The use of `Select` projects the results into an anonymous type, so `var` is required.
- **Displaying the results:** It prints a formatted table to the console. `{0, 5}` means right-align `ProductId` in 5 spaces. `{1, -35}` means left-align `ProductName` in 35 spaces. `{2, 8:C}` means format `UnitPrice` as currency in 8 spaces (culture-sensitive, so in US systems, it'll show \$).
- **Logging:** After the table, it prints the SQL query generated by EF Core. This is useful for debugging. Then it prints connection info for verification.

1. Run the console app, enter a unit price of 60, and note the results include five products and the SQL that was executed by SQL Server, as shown in

the following partial output:

```
Enter a unit price: 60
```

```
-----  
| Id | Name | Price |  
-----  
| 9 | Mishi Kobe Niku | £97.00 | | 18 |  
Carnarvon Tigers | £62.50 | | 20 | Sir  
Rodney's Marmalade | £81.00 | | 29 |  
Thüringer Rostbratwurst | £123.79 | | 38 |  
Côte de Blaye | £263.50 |  
-----
```

```
DECLARE @price money = 60.0; SELECT [p].  
[ProductId], [p].[ProductName], [p].  
[UnitPrice] FROM [Products] AS [p] WHERE  
[p].[UnitPrice] > @price Provider:  
Microsoft.EntityFrameworkCore.SqlServer  
Connection: Data  
Source=tcp:127.0.0.1,1433;Initial  
Catalog=Northwind;Persist Security  
Info=False;User  
ID=sa;Password=<censored>;Multiple Active  
Result Sets=False;Encrypt=True;Trust  
Server Certificate=False;Connection  
Timeout=10;
```

Calculated properties on entity creation

EF Core 7 added an `IMaterializationInterceptor` interface that allows interception before and after an entity is created, and when properties are initialized. This is useful for calculated values.

For example, when a service or client app requests entities to show to the user, it might want to cache a copy of the entity for a period of time. To do this, it needs

to know when the entity was last refreshed. It would be useful if this information were automatically generated and stored with each entity at the time of loading.

To achieve this goal, we must complete four steps:

1. First, define an interface with the extra property.
2. Next, at least one entity model class must implement the interface.
3. Then, define a class that implements the interceptor interface with a method named `InitializedInstance` that will execute on any entity, and if that entity implements the custom interface with the extra property, then it will set its value.
4. Finally, we must create an instance of the interceptor and register it in the data context class.

Now let's implement this for Northwind `Employee` entities:

1. In the `Northwind.EntityModels` project, add a new file named `IHasLastRefreshed.cs`, and modify its contents to define the interface, as shown in the following code:

```
namespace Northwind.EntityModels; public
interface IHasLastRefreshed {
    DateTimeOffset LastRefreshed { get; set; }
}
```

2. In the `Northwind.EntityModels` project, add a new file named `EmployeePartial.cs`, and modify its contents to implement the interface, as shown highlighted in the following code:

```
using
System.ComponentModel.DataAnnotations.Schema;
// To use [NotMapped]. namespace
Northwind.EntityModels; public partial
class Employee : IHasLastRefreshed {
```



```
[NotMapped] public DateTimeOffset  
LastRefreshed { get; set; } }
```

Good practice: Add extension code like this to a separate partial entity class file so that you can later regenerate the Employee.cs file using the `dotnet-ef` tool without overwriting your additional code.

1. In the Northwind.DataContext project, add a new file named SetLastRefreshedInterceptor.cs, and modify its contents to define the interceptor, as shown in the following code:

```
// IMaterializationInterceptor,  
MaterializationInterceptionData using  
Microsoft.EntityFrameworkCore.Diagnostics;  
namespace Northwind.EntityModels; public  
class SetLastRefreshedInterceptor :  
IMaterializationInterceptor { public  
object InitializedInstance(  
MaterializationInterceptionData  
materializationData, object entity) { if  
(entity is IHasLastRefreshed  
entityWithLastRefreshed) {  
entityWithLastRefreshed.LastRefreshed =  
DateTimeOffset.UtcNow; } return entity; }  
}
```

2. In the Northwind.DataContext project, in NorthwindContext.cs, delete the existing OnConfiguring method.

3. In the `Northwind.DataContext` project, add a new file named `NorthwindContextPartial.cs`, then declare and register the interceptor in the `OnConfiguring` method, as shown in the following code:

```
using Microsoft.Data.SqlClient; // To use
SqlConnectionStringBuilder. using
Microsoft.EntityFrameworkCore; // To use
DbContext. namespace
Northwind.EntityModels; public partial
class NorthwindContext : DbContext {
private static readonly
SetLastRefreshedInterceptor
setLastRefreshedInterceptor = new();
protected override void
OnConfiguring(DbContextOptionsBuilder
optionsBuilder) { if (!
optionsBuilder.IsConfigured) {
SqlConnectionStringBuilder builder =
new(); builder.DataSource =
"tcp:127.0.0.1,1433"; // SQL Server in
container. builder.InitialCatalog =
"Northwind";
builder.TrustServerCertificate = true;
builder.MultipleActiveResultSets = true;
// Because we want to fail fast. Default
is 15 seconds. builder.ConnectTimeout = 3;
// SQL Server authentication.
builder.UserID =
Environment.GetEnvironmentVariable("MY_SQL_USR");
builder.Password =
Environment.GetEnvironmentVariable("MY_SQL_PWD");
optionsBuilder.UseSqlServer(builder.ConnectionString);
optionsBuilder.LogTo(NorthwindContextLogger.Write
[
```

```
Microsoft.EntityFrameworkCore.Diagnostics
.RelationalEventId.CommandExecuting ]); }
optionsBuilder.AddInterceptors(setLastRefreshedIn
} }
```

4. Save the changes.

Controlling the tracking of entities

We need to start with the definition of entity **identity resolution**. EF Core resolves each entity instance by reading its unique primary key value. This ensures there are no ambiguities about the identities of entities or relationships between them.

EF Core can only track entities with keys because it uses a key to uniquely identify the entity in the database. Keyless entities, like those returned by views, are never tracked.

By default, EF Core assumes that you want to track entities in local memory so that if you make changes, like adding a new entity, modifying an existing entity, or removing an existing entity, then you can call `SaveChanges` and all those changes will be made in the underlying data store.

If you execute a query within a data context, like getting all customers in Germany, and then execute another query within the same data context, like getting all customers whose name starts with A, if one of those customer entities already exists in the context, it will be identified and not replaced or loaded twice. However, if the telephone number of that customer is updated in the database between the executions of the two queries, then the entity being tracked in the data context is not refreshed with the new telephone number.

If you do not need to track these changes, or you want to load new instances of an entity for every query execution with the latest data values, even if the entity

is already loaded, then you can disable tracking.

To disable tracking for an individual query, call the `AsNoTracking` method as part of the query, as shown in the following code:

```
var products = db.Products .AsNoTracking()  
    .Where(p => p.UnitPrice > price);
```

Warning! You can only track queries that return whole entities, like the preceding query that returns whole `Product` instances. If you query projects into an anonymous type, for example, by appending the following to the preceding query: `.Select (p => new { p.ProductId, p.ProductName, p.UnitPrice })`, then it cannot be tracked!

To disable tracking by default for the data context, set the change tracker's query tracking behavior to `NoTracking`, as shown in the following code:

```
db.ChangeTracker.QueryTrackingBehavior =  
    QueryTrackingBehavior.NoTracking;
```

To disable tracking for an individual query, but retain identity resolution, call the `AsNoTrackingWithIdentityResolution` method as part of the query, as shown in the following code:

```
var products = db.Products  
    .AsNoTrackingWithIdentityResolution() .Where(p  
=> p.UnitPrice > price);
```

To disable tracking but perform identity resolution by default for the data context, set the change tracker's query-tracking behavior to `NoTrackingWithIdentityResolution`, as shown in the following code:

```
db.ChangeTracker.QueryTrackingBehavior =  
QueryTrackingBehavior.NoTrackingWithIdentityResolution
```

To set defaults for all new instances of a data context, in the `OnConfiguring` method, call the `UseQueryTrackingBehavior` method, as shown in the following code:

```
protected override void OnConfiguring(  
    DbContextOptionsBuilder optionsBuilder) {  
    optionsBuilder.UseSqlServer(connectionString)  
        .UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking);  
}
```

To understand how tracking works in detail, we will next review three scenarios:

- Default tracking
- No tracking
- No tracking with identity resolution

In each of the three scenarios, we will perform the same actions and note the different behavior:

1. Run a query for customers in Germany that loads them into the data context.
2. Change the telephone number for one of the customers *in the database*.
3. Run a query for customers in Germany or whose company name starts with the letter A. This may or may not refresh any existing entities in the data context.

- 4. Change the telephone number for one of the customers *in the data context*.
- 5. Update the database with any tracked changes.

A scenario using default tracking

The default is **change tracking** with identity resolution. Once an entity is loaded into the data context, underlying changes are not reflected, and only one copy exists. Entities have local changes tracked and a call to `SaveChanges` updates the database, as shown in *Table 8.2*:

Entities in database context				
Alfred's Futterkiste	123-4567	any		
Alfred's Futterkiste	123-8905			
Alfred's Futterkiste	123-9876	with A		
Alfred's Futterkiste	123-9876	any		
Alfred's Futterkiste	123-9876			
Alfred's Futterkiste	123-9876			
Alfred's Futterkiste	123-1928			

Table 8.2: Default tracking scenario

The same scenario using no tracking

No tracking and no identity resolution. Every query loads another instance of a database row into the data context, including underlying changes, allowing duplicates and mixed out-of-date and updated data. No local entity changes are tracked, so `SaveChanges` does nothing, as shown in *Table 8.3*:

Entities in database context				
Alfred's Futterkiste	123-4567	any		
Alfred's Futterkiste	123-8905			
Alfred's Futterkiste	123-9876	with A		
Alfred's Futterkiste	123-9876			
Alfred's Futterkiste	123-9876			
Alfred's Futterkiste	123-9876			
Alfred's Futterkiste	123-1928			

	Alfred's Futterkiste, 123-9876		
	Alfred's Futterkiste, 123-9876		
	Alfred's Futterkiste, 123-9876		
	Alfred's Futterkiste, 123-9876		
	Alfred's Futterkiste, 123-1928		
	Alfred's Futterkiste, 123-9876		
	Alfred's Futterkiste, 123-9876		
	Alfred's Futterkiste, 123-1928		

Table 8.3: No tracking scenario

The same scenario using no tracking with identity resolution

No tracking with identity resolution. Once an entity is loaded into the data context, underlying changes are not reflected, and only one copy exists. No local entity changes are tracked, so `SaveChanges` does nothing, as shown in Table 8.4:

Entity in database context			
Alfred's Futterkiste, 123-9876			
Alfred's Futterkiste, 123-9876			
Alfred's Futterkiste, 123-9876 with A			
Alfred's Futterkiste, 123-9876			
Alfred's Futterkiste, 123-9876			
Alfred's Futterkiste, 123-9876			
Alfred's Futterkiste, 123-9876			

Table 8.4: Identity resolution scenario

Summary of tracking

Which should you choose? Of course, it depends on your specific scenario. You will sometimes read blogs that excitedly tell you that you can dramatically

improve your EF Core queries by calling `AsNoTracking`. But if you run a query that returns thousands of entities and then run the same query again within the same data context, you now have thousands of duplicates! This wastes memory and impacts performance.

Understand how the three tracking choices work and select the best one for your data context or individual queries. In the next topic, you will learn how to map inheritance hierarchies.

Mapping inheritance hierarchies with EF Core

Imagine that you have an inheritance hierarchy for some C# classes to store information about students and employees, both of which are types of people. All people have a name and an ID to uniquely identify them, students have a subject they are studying, and employees have a hire date, as shown in the following code:

```
public abstract class Person { public int Id { get; set; } public required string Name { get; set; } } public class Student : Person { public required string Subject { get; set; } } public class Employee : Person { public DateTime HireDate { get; set; } }
```

By default, EF Core will map these to a single table using the **table-per-hierarchy (TPH)** mapping strategy. EF Core 5 introduced support for the **table-per-type (TPT)** mapping strategy. EF Core 7 introduced support for the **table-per-concrete-type (TPC)** mapping strategy. Let's explore the differences between these mapping strategies.

Table-per-hierarchy (TPH) mapping

strategy

For the Person-Student-Employee hierarchy, TPH will use a single table structure with a discriminator column to indicate which type of person, a student or employee, the row is, with nullable columns for extra values that only apply to some of the types, as shown highlighted in the following code:

```
CREATE TABLE [People] ( [Id] int NOT NULL
IDENTITY, [Name] nvarchar(max) NOT NULL,
[Discriminator] nvarchar(max) NOT NULL,
[Subject] nvarchar(max) NULL, [HireDate]
nvarchar(max) NULL, CONSTRAINT [PK_People]
PRIMARY KEY ([Id]) );
```

Some data in the table might look like *Table 8.5*:

Person									
Student									
Employee									
Student									
Employee									

Table 8.5: Sample data in the People table

TPH requires the `Discriminator` column to store the class name of the type for each row. TPH requires the columns for properties of derived types to be nullable, like `Subject` and `HireDate`. This will cause an issue if those properties are required (non-null) at the class level. EF Core does not handle this by default.

The main benefits of the TPH mapping strategy are simplicity and performance, which is why it is used by default.

Good practice: If the discriminator column has many different values, then you can improve

performance even more by defining an index on the discriminator. But if there are only a few different values, an index may make overall performance worse because it affects updating time. In this case, there are only two potential values, Student and Employee, so in a table with 100,000 rows, an index would make little difference.

Table-per-type (TPT) mapping strategy

For the Person-Student-Employee hierarchy, TPT will use a table for every type, as shown in the following code:

```
CREATE TABLE [People] ( [Id] int NOT NULL
IDENTITY, [Name] nvarchar(max) NOT NULL,
CONSTRAINT [PK_People] PRIMARY KEY ([Id]) );
CREATE TABLE [Students] ( [Id] int NOT NULL,
[Subject] nvarchar(max) NULL, CONSTRAINT
[PK_Students] PRIMARY KEY ([Id]) CONSTRAINT
[FK_Students_People] FOREIGN KEY ([Id])
REFERENCES [People] ([Id]) ); CREATE TABLE
[Employees] ( [Id] int NOT NULL, [HireDate]
nvarchar(max) NULL, CONSTRAINT [PK_Employees]
PRIMARY KEY ([Id]) CONSTRAINT
[FK_Employees_People] FOREIGN KEY ([Id])
REFERENCES [People] ([Id]) );
```

Some data in the tables might look like the following:

Name		
Roman Roy		
Kendall Roy		
Stobhan Roy		

Table 8.6: People table

Subject		
History		

Table 8.7: Students table

HireDate		
02/04/2014		
12/09/2020		

Table 8.8: Employees table

The main benefit of the TPT mapping strategy is reduced storage due to the full normalization of the data. The main disadvantage is that a single entity is spread over multiple tables, and reconstructing it takes more effort and therefore reduces overall performance. TPT is usually a poor choice, so only use it if the table structure is already normalized and cannot be restructured.

Table-per-concrete-type (TPC) mapping strategy

For the Person-Student-Employee hierarchy, TPC will use a table for each non-abstract type, as shown in the following code:

```
CREATE TABLE [Students] ( [Id] int NOT NULL
DEFAULT (NEXT VALUE FOR [PersonIds]), [Name]
nvarchar(max) NOT NULL, [Subject]
nvarchar(max) NULL, CONSTRAINT [PK_Students]
PRIMARY KEY ([Id]) CONSTRAINT
[FK_Students_People] FOREIGN KEY ([Id])
REFERENCES [People] ([Id]) ); CREATE TABLE
[Employees] ( [Id] int NOT NULL DEFAULT (NEXT
VALUE FOR [PersonIds]), [Name] nvarchar(max)
NOT NULL, [HireDate] nvarchar(max) NULL,
```

```
CONSTRAINT [PK_Employees] PRIMARY KEY ([Id])
CONSTRAINT [FK_Employees_People] FOREIGN KEY
([Id]) REFERENCES [People] ([Id]) );
```

Since there is not a single table with an IDENTITY column to assign Id values, we can use the (NEXT VALUE FOR [PersonIds]) command to define a sequence shared between the two tables so they do not assign the same Id values.

Some data in the tables might look like the following:

Subject				
History Roy				

Table 8.9: Students table

HireDate				
2014-11-04				
2014-11-04				
2014-11-04				

Table 8.10: Employees table

The main benefit of the TPC mapping strategy is performance because when querying a single concrete type, only one table is needed, so we avoid expensive joins. It works best for large inheritance hierarchies of many concrete types, each with many type-specific properties.

Configuring inheritance hierarchy mapping strategies

First, all types must be included in the model, as shown in the following code:

```
public DbSet<Person> People { get; set; }
public DbSet<Student> Students { get; set; }
public DbSet<Employee> Employees { get; set; }
```

For TPH, you are now finished, because it is the default! If you want to make this explicit, then in the data context class `OnModelCreating` method, call the appropriate “use mapping strategy” method on the base class of the hierarchy. `Person` is the base class, so you would call `UseTphMappingStrategy` on that entity type, as shown in the following code:

```
modelBuilder.Entity<Person>().UseTphMappingStrategy();
```

To use either of the other two mapping strategies, call the appropriate method, as shown in the following code:

```
modelBuilder.Entity<Person>().UseTptMappingStrategy();
modelBuilder.Entity<Person>().UseTpcMappingStrategy();
```

Next, you can optionally specify the table name to use for each entity class, as shown in the following code:

```
modelBuilder.Entity<Student>().ToTable("Students");
modelBuilder.Entity<Employee>().ToTable("Employees");
```

The TPC strategy should have a shared sequence, so we should configure that too, as shown in the following code:

```
modelBuilder.HasSequence<int>("PersonIds");
modelBuilder.Entity<Person>().UseTpcMappingStrategy()
    .Property(e => e.Id).HasDefaultValueSql("NEXT
```

```
VALUE FOR [PersonIds]");
```

Example of hierarchy mapping strategies

Now let's see this in action using a new database and project named HierarchyMapping:

1. Use your preferred code editor to add a console app project, as defined in the following list:
 - Project template: **Console App** / console
 - Solution file and folder: ModernApps
 - Project file and folder:
Northwind.Console.HierarchyMapping
 - **Do not use top-level statements**: Cleared
 - **Enable native AOT publish**: Cleared
7. Configure the startup project to run
Northwind.Console.HierarchyMapping.
8. In the Northwind.Console.HierarchyMapping project, add package references to the EF Core data provider for SQL Server, and globally and statically import the System.Console class, as shown in the following markup:

```
<ItemGroup Label="Versions are set in  
Directory.Build.props."> <PackageReference  
Include="Microsoft.EntityFrameworkCore.Design"  
> <PackageReference  
Include="Microsoft.EntityFrameworkCore.SqlServer"  
> </ItemGroup> <ItemGroup Label="To  
simplify use of WriteLine."> <Using  
Include="System.Console" Static="true" >  
</ItemGroup>
```

9. Build the project to restore packages.
10. In the `Northwind.Console.HierarchyMapping` project, add a new folder named `Models`.
11. In `Models`, add a new class file named `Person.cs`, and modify its contents, as shown in the following code:

```
using
System.ComponentModel.DataAnnotations; //
To use [Required]. namespace
Northwind.Models; public abstract class
Person { public int Id { get; set; }
[Required] [StringLength(40)] public
required string Name { get; set; } }
```

12. In `Models`, add a new class file named `Student.cs`, and modify its contents, as shown in the following code:

```
namespace Northwind.Models; public class
Student : Person { public required string
Subject { get; set; } }
```

13. In `Models`, add a new class file named `Employee.cs`, and modify its contents, as shown in the following code:

```
namespace Northwind.Models; public class
Employee : Person { public DateTime
HireDate { get; set; } }
```

14. In `Models`, add a new class file named `HierarchyDb.cs`, and modify its contents, as shown in the following code:

```

using Microsoft.EntityFrameworkCore; // To
use DbSet<T>. namespace Northwind.Models;
public class HierarchyDb : DbContext {
    public DbSet<Person>? People { get; set; }
    public DbSet<Student>? Students { get;
    set; } public DbSet<Employee>? Employees {
    get; set; } public
HierarchyDb(DbContextOptions<HierarchyDb>
options) : base(options) { } protected
override void OnModelCreating(ModelBuilder
modelBuilder) {
    modelBuilder.Entity<Person>()
        .UseTphMappingStrategy(); // Populate
        database with sample data. Student p1 =
        new() { Id = 1, Name = "Roman Roy",
        Subject = "History" }; Employee p2 = new()
        { Id = 2, Name = "Kendall Roy", HireDate =
        new(year: 2014, month: 4, day: 2) };
        Employee p3 = new() { Id = 3, Name =
        "Siobhan Roy", HireDate = new(year: 2020,
        month: 9, day: 12) };
        modelBuilder.Entity<Student>().HasData(p1);
        modelBuilder.Entity<Employee>().HasData(p2,
        p3); } }

```

15. In Program.cs, delete the existing statements. Then add statements to configure the connection string for the HierarchyDb data context, and use it to delete and create a database named HierarchyMapping (not Northwind!). After that, show the automatically generated SQL script, and output the students, employees, and people, as shown in the following code:

```

using Microsoft.Data.SqlClient; // To use
SqlConnectionStringBuilder. using

```



```

Microsoft.EntityFrameworkCore; //
GenerateCreateScript() using
Northwind.Models; // To use HierarchyDb,
Person, Student, Employee.
DbContextOptionsBuilder<HierarchyDb>
options = new();
SqlConnectionStringBuilder builder =
new(); builder.DataSource =
"tcp:127.0.0.1,1433";
builder.InitialCatalog =
"HierarchyMapping";
builder.TrustServerCertificate = true;
builder.MultipleActiveResultSets = true;
// Because we want to fail faster. Default
is 15 seconds. builder.ConnectTimeout = 3;
// If using Windows Integrated
authentication. //
builder.IntegratedSecurity = true; // If
using SQL Server authentication.
builder.UserID =
Environment.GetEnvironmentVariable("MY_SQL_USR");
builder.Password =
Environment.GetEnvironmentVariable("MY_SQL_PWD");
options.UseSqlServer(builder.ConnectionString);
using (HierarchyDb db =
new(options.Options)) { bool deleted =
await db.Database.EnsureDeletedAsync();
WriteLine($"Database deleted: {deleted}");
bool created = await
db.Database.EnsureCreatedAsync();
WriteLine($"Database created: {created}");
WriteLine("SQL script used to create the
database:");
WriteLine(db.Database.GenerateCreateScript());
if (db.Students is null || !

```

```

db.Students.Any()) { WriteLine("There are
no students."); } else { foreach (Student
student in db.Students) { WriteLine("{0}
studies {1}", student.Name,
student.Subject); } } if (db.Employees is
null || !db.Employees.Any()) {
WriteLine("There are no employees."); }
else { foreach (Employee employee in
db.Employees) { WriteLine("{0} was hired
on {1}", employee.Name,
employee.HireDate); } } if (db.People is
null || !db.People.Any()) {
WriteLine("There are no people."); } else
{ foreach (Person person in db.People) {
WriteLine("{0} has ID of {1}",
person.Name, person.Id); } } }

```

16. Start the console app, and note the results including the single table named `People` that is created, as shown in the following output:

```

Database deleted: False Database created:
True SQL script used to create the
database: CREATE TABLE [People] ( [Id] int
NOT NULL IDENTITY, [Name] nvarchar(40) NOT
NULL, [Discriminator] nvarchar(8) NOT
NULL, [HireDate] datetime2 NULL, [Subject]
nvarchar(max) NULL, CONSTRAINT [PK_People]
PRIMARY KEY ([Id]) ); GO IF EXISTS (SELECT
* FROM [sys].[identity_columns] WHERE
[name] IN (N'Id', N'Discriminator',
N'Name', N'Subject') AND [object_id] =
OBJECT_ID(N'[People]')) SET
IDENTITY_INSERT [People] ON; INSERT INTO
[People] ([Id], [Discriminator], [Name],

```

```

[Subject]) VALUES (1, N'Student', N'Roman
Roy', N'History'); IF EXISTS (SELECT *
FROM [sys].[identity_columns] WHERE [name]
IN (N'Id', N'Discriminator', N'Name',
N'Subject') AND [object_id] =
OBJECT_ID(N'[People]')) SET
IDENTITY_INSERT [People] OFF; GO IF EXISTS
(SELECT * FROM [sys].[identity_columns]
WHERE [name] IN (N'Id', N'Discriminator',
N'HireDate', N'Name') AND [object_id] =
OBJECT_ID(N'[People]')) SET
IDENTITY_INSERT [People] ON; INSERT INTO
[People] ([Id], [Discriminator],
[HireDate], [Name]) VALUES (2,
N'Employee',
'2014-04-02T00:00:00.0000000', N'Kendall
Roy'), (3, N'Employee',
'2020-09-12T00:00:00.0000000', N'Siobhan
Roy'); IF EXISTS (SELECT * FROM [sys].
[identity_columns] WHERE [name] IN (N'Id',
N'Discriminator', N'HireDate', N'Name')
AND [object_id] = OBJECT_ID(N'[People]'))
SET IDENTITY_INSERT [People] OFF; GO Roman
Roy studies History Kendall Roy was hired
on 02/04/2014 00:00:00 Siobhan Roy was
hired on 12/09/2020 00:00:00 Roman Roy has
ID of 1 Kendall Roy has ID of 2 Siobhan
Roy has ID of 3

```

17. In your preferred database tool, view the contents of the `People` table, as shown in *Figure 8.2*:

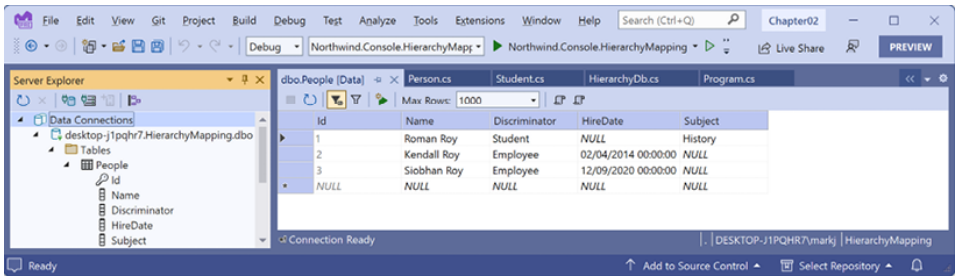


Figure 8.2: The People table when using the TPH mapping strategy

1. Close the connection to the HierarchyMapping database.
2. In HierarchyDb.cs, comment out the method call that configures TPH and add a call to the method that configures TPT, as shown highlighted in the following code:

```
protected override void
OnModelCreating(ModelBuilder modelBuilder)
{ modelBuilder.Entity<Person>() //
.UseTphMappingStrategy();
.UseTptMappingStrategy();
```

3. Start the console app, and note the results, including the three tables named People, Students, and Employees that are created, as shown in the following partial output:

```
Database deleted: True Database created:
True SQL script used to create the
database: CREATE TABLE [People] ( [Id] int
NOT NULL IDENTITY, [Name] nvarchar(40) NOT
NULL, CONSTRAINT [PK_People] PRIMARY KEY
([Id]) ); GO CREATE TABLE [Employees] (
[Id] int NOT NULL, [HireDate] datetime2
NOT NULL, CONSTRAINT [PK_Employees]
PRIMARY KEY ([Id]), CONSTRAINT
```

```
[FK_Employees_People_Id] FOREIGN KEY
([Id]) REFERENCES [People] ([Id]) ); GO
CREATE TABLE [Students] ( [Id] int NOT
NULL, [Subject] nvarchar(max) NULL,
CONSTRAINT [PK_Students] PRIMARY KEY
([Id]), CONSTRAINT [FK_Students_People_Id]
FOREIGN KEY ([Id]) REFERENCES [People]
([Id]) ); GO
```

4. In your preferred database tool, view the contents of the tables, as shown in *Figure 8.3*:

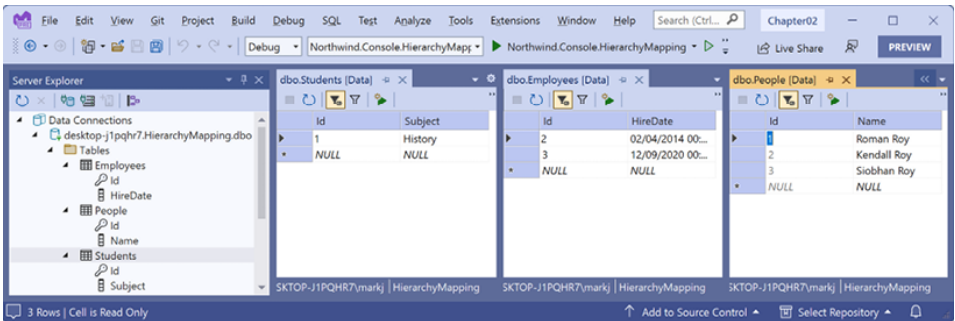


Figure 8.3: The tables when using the TPT mapping strategy

1. Close the connection to the `HierarchyMapping` database.
2. In `HierarchyDb.cs`, comment out the method call that configures TPT. Then add a call to the method that configures TPC, and configure a sequence to track assigned ID values starting at a value of `4` because we always add three sample rows, as shown highlighted in the following code:

```
protected override void
OnModelCreating(ModelBuilder modelBuilder)
{ modelBuilder.Entity<Person>() //
.UseTphMappingStrategy(); //
.UseTptMappingStrategy(); }
```

```

.UseTpcMappingStrategy() .Property(person
=> person.Id) .HasDefaultValueSql("NEXT
VALUE FOR [PersonIds]");
modelBuilder.HasSequence<int>("PersonIds",
builder => { builder.StartsAt(4); });

```

3. Start the console app, and note the results including the two tables named `Students` and `Employees` that are created as well as the shared sequence that starts at 4, as shown in the following partial output:

```

CREATE SEQUENCE [PersonIds] AS int START
WITH 4 INCREMENT BY 1 NO MINVALUE NO
MAXVALUE NO CYCLE; GO CREATE TABLE
[Employees] ( [Id] int NOT NULL DEFAULT
(NEXT VALUE FOR [PersonIds]), [Name]
nvarchar(40) NOT NULL, [HireDate]
datetime2 NOT NULL, CONSTRAINT
[PK_Employees] PRIMARY KEY ([Id]) ); GO
CREATE TABLE [Students] ( [Id] int NOT
NULL DEFAULT (NEXT VALUE FOR [PersonIds]),
[Name] nvarchar(40) NOT NULL, [Subject]
nvarchar(max) NULL, CONSTRAINT
[PK_Students] PRIMARY KEY ([Id]) ); GO

```

4. In your preferred database tool, view the contents of the tables, as shown in *Figure 8.4*:

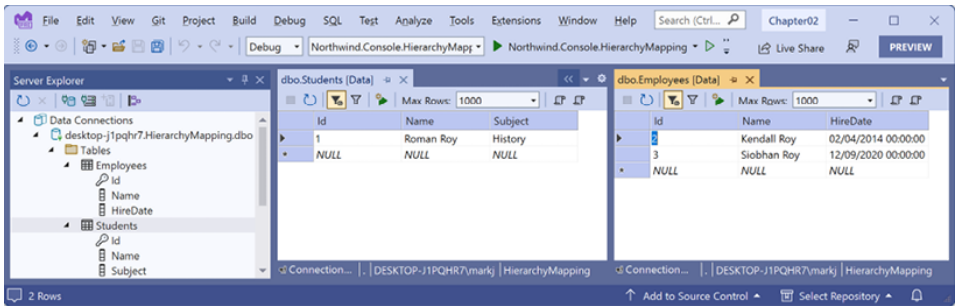


Figure 8.4: The tables when using the TPC mapping strategy

1. Close the connection to the HierarchyMapping database.
2. In Program.cs, after the statement to write the database create script to the console, add some statements to add two new people using the current database context, as shown highlighted in the following code:

```
WriteLine(db.Database.GenerateCreateScript());
if ((db.Employees is not null) &&
    (db.Students is not null)) {
    db.Students.Add(new Student { Name =
        "Connor Roy", Subject = "Politics" });
    db.Employees.Add(new Employee { Name =
        "Kerry Castellabate", HireDate =
        DateTime.UtcNow }); int result =
    db.SaveChanges(); WriteLine($"{result}
    people added."); }
```

3. Start the console app, and note the results, including the two new people added using the database context with IDs that start at 4, as shown in the following partial output:

```
2 people added. Roman Roy studies History
Connor Roy studies Politics Kendall Roy
```

```
was hired on 02/04/2014 00:00:00 Siobhan  
Roy was hired on 12/09/2020 00:00:00 Kerry  
Castellabate was hired on 19/05/2023  
10:13:53 Kendall Roy has ID of 2 Siobhan  
Roy has ID of 3 Kerry Castellabate has ID  
of 4 Roman Roy has ID of 1 Connor Roy has  
ID of 5
```

You've now seen how an object-relational mapper like EF Core can define an object inheritance hierarchy and map it in three different ways to an underlying database structure with one or more related tables. You've also seen how Code First works especially well with this, since it is so easy to delete and recreate the database each time the project starts.

Practicing and exploring

Test your knowledge and understanding by answering some questions, getting some hands-on practice, and exploring this chapter's topics with deeper research.

Exercise 8.1 – Online material

Online material can be extra content written by me for this book, or it can be references to content created by Microsoft or third parties.

Review performance choices

The data tier can have an outsized influence on the overall performance of an app or service.

Docs: <https://learn.microsoft.com/en-us/ef/core/performance/>.

Exercise 8.2 – Practice exercises

Benchmarking ADO.NET against EF Core

In the `ModernApps` solution, create a console app named `Exercise_ADONETvsEFCore` that uses `Benchmark.NET` to compare retrieving all the products from the Northwind database using ADO.NET (`SqlClient`) and using EF Core.

You can learn how to use `Benchmark.NET` by reading the online-only section *Benchmarking Performance and Testing* at the following link:
<https://github.com/markjprice/appsservices-net10/blob/main/docs/ch01-benchmarking.md>.

Exercise 8.3 – Test your knowledge

Answer the following questions:

1. What can the `dotnet-ef` tool be used for?
2. What type would you use for the property that represents a table, for example, the `Products` property of a data context?
3. What type would you use for the property that represents a one-to-many relationship, for example, the `Products` property of a `Category` entity?
4. What is the EF Core convention for primary keys?
5. Why might you choose the Fluent API in preference to annotation attributes?
6. Why might you implement the `IMaterializationInterceptor` interface in an entity type?

Exercise 8.4 – Explore topics

Use the links on the following page to learn more details about the topics covered in this chapter: <https://github.com/markjprice/apps-services-net10/blob/main/docs/book-links.md#chapter-8---building-entity-models-using-ef-core>.

Summary

In this chapter, you learned:

- How to execute a simple query and process the results using the slower but more object-oriented EF Core.
- How to configure and decide between three mapping strategies for type hierarchies.
- How to implement calculated properties on entity creation.

In the next chapter, you will learn how to build an LLM-based chat service that will fine-tune a standard model with custom information.

Get This Book

UNLOCK NOW

ion and

Exclusive

Scan the QR code

book by name, co



Search for this

ps on the page.

Note: Keep your invoice handy. Purchases made directly from Packt don't

require one.

9

Building an LLM-Based Chat Service

Integrating a **large language model (LLM)** like OpenAI's GPT-5.2 into a .NET project can significantly enhance its capabilities, offering advanced natural language processing, content generation, and more.

This chapter is about building an LLM-based chat service that will fine-tune a standard model with custom information, including PDF and Markdown files, a biography of this book's author, and information from the Northwind database. This allows the user to ask questions about the author and a fictional corporation's database.

You will also learn how to build a **Model Context Protocol (MCP)** server which is the latest standard way to extend the capabilities of an AI agent. Finally, you will learn how to use local models for no-cost intelligence.

At the time of writing, cloud-based LLM APIs cost money. In this chapter, we will use one of the least expensive options: OpenAI's GPT-5 nano. As OpenAI says, "Our frontier models are designed to spend more time thinking before producing a response, making them ideal for complex, multi-

step problems. GPT-5 nano is the fastest, most cost-efficient version of GPT-5.” The input cost is US \$0.05 / 1 M tokens. The output cost is US\$0.40 / 1 M tokens. When I completed the tasks while writing this chapter, it cost me less than US \$0.03.

This chapter covers the following topics:

- Introducing LLMs
- Using Agent Framework with an OpenAI model
- Building an MCP server
- Running local LLMs

Introducing LLMs

The terms **LLM** and **GPT (Generative Pre-trained Transformer)** are often used interchangeably, but they’ve got their nuances:

- LLM is a more general term that refers to any language model with a large number (in the billions) of parameters that has been trained on vast amounts of text data and can perform a variety of language-related tasks, such as translation, summarization, question-answering, and more.
- GPT is a series of models developed by OpenAI. It’s like saying iPhone versus smartphones; the iPhone is a type of smartphone. Similarly, GPT is a type of LLM. The “transformer” part refers to the architecture they’re built on, a groundbreaking model structure introduced in 2017 that’s particularly good at handling sequences of data, like sentences in a text. While all GPT models are based on the Transformer architecture, not all LLMs must be.

So, every GPT is an LLM, but not every LLM is a GPT.

LLMs are used in various applications, including:

- **Chatbots and virtual assistants:** Providing human-like interactions and support, for example, ChatGPT 5.2 or Gemini 3 and Microsoft Copilot in Windows and Office.
- **Content creation:** Assisting in writing articles, reports, and even creative writing, for example, Microsoft Designer: <https://designer.microsoft.com/>.
- **Translation:** Offering language translation services.
- **Coding assistance:** Helping developers with code suggestions and debugging, for example, GitHub Copilot and JetBrains AI Assistant.

How LLMs work

The most important concepts to understand about LLMs include the following:

LLMs are trained on vast amounts of text data sourced from books, websites like Stack Overflow and Reddit, articles, and other textual content. This extensive dataset provides the model with a broad understanding of language, context, and factual knowledge up until the training cutoff.

You can find more information on Stack Overflow and OpenAI's partnership through the following link <https://stackoverflow.co/company/press/archive/openai-partnership>. Similarly, the following link contains more information regarding OpenAI and Reddit's partnership: <https://openai.com/index/openai-and-reddit-partnership/>.

Some LLMs use a type of neural network architecture called a **Transformer**. The Transformer architecture consists of multiple layers of attention mechanisms and feed-forward networks:

- **Attention mechanisms:** These allow the model to weigh the importance

of different words in a sentence, helping it focus on relevant parts of the input when generating responses.

- **Feed-forward neural networks:** These process the weighted inputs from the attention mechanisms and generate the output for each layer.

Interestingly, the current “dumbness” of LLMs can be attributed to this feed-forward-only nature. Experiments with building systems that feed back into themselves may lead to higher-level cognition because that allows for self-reflection and continuous learning.

Text input is divided into smaller units called tokens. Tokens can be words, subwords, or even characters. Tokenization helps the model handle large vocabularies efficiently. For example, the sentence “Hello, world!” might be tokenized into ["Hello", ",", "world", "!"].

The training process involves:

- **Forward pass:** The input text is passed through the model, which generates a prediction. For instance, given a sentence fragment, the model predicts the next word.
- **Loss calculation:** The model’s prediction is compared to the actual next word in the training data, and a loss value is calculated. This loss measures the difference between the prediction and the actual word.
- **Backward pass and optimization:** Using techniques like backpropagation and gradient descent, the model adjusts its internal parameters to minimize the loss. This process is repeated for many iterations across the training dataset.

After the initial training, models are often fine-tuned on specific datasets to improve their performance in particular domains or tasks. Fine-tuning involves further training the model on a smaller, more targeted dataset. For example, Packt could fine-tune a custom LLM with all the content from all the books they

publish and then set a custom chatbot loose in their Discord channels. It would hopefully then be able to answer questions about any of Packt's books in a knowledgeable way, including helping readers if they get stuck on a particular page in a section.

During inference, aka usage, the trained model generates text based on the input it receives. This involves:

- **Contextual understanding:** The model uses its trained knowledge to understand the context of the input text.
- **Prediction:** The model predicts the most likely next token or sequence of tokens, generating coherent and contextually appropriate text.

LLMs can manage ambiguity and maintain context over long passages of text due to their attention mechanisms. They can recall relevant information from earlier parts of the conversation, enabling them to generate coherent and contextually accurate responses. The context windows of models are constantly growing.

Despite their capabilities, LLMs have limitations:

- **Bias:** Models can inherit biases present in the training data, leading to biased or unfair outputs.
- **Factual inaccuracies:** While they generate human-like text, they can also produce incorrect or nonsensical information. This is especially dangerous because they are so creative and produce realistic-sounding legal advice and other advice. Treat LLMs as junior team members who always need their work checked.
- **Dependency on data:** Their knowledge is limited to the data they were trained on, which may become outdated.

Now that you understand how LLMs work and some of their benefits and limitations, let's find out how to get access to one.

Obtaining access to an LLM

We need access to an LLM for our chat system. The easiest way to get access to a cloud-based LLM suitable for integration with .NET projects is via either OpenAI or Azure OpenAI. At the time of writing, applying for an Azure OpenAI model requires a business email address. Gmail or MSN accounts are not accepted. For this reason, this chapter will primarily use OpenAI.

If you do have a business email, and you would prefer to pay to use Azure, then you can learn how to create an Azure OpenAI model at the following link: <https://learn.microsoft.com/azure/ai-services/openai/how-to/create-resource>.

You'll need two pieces of information because you cannot use a cloud-based model without them:

- **API key:** This is given in the portal for your LLM service and should be kept secret. Anyone with your API key can use your account resources.
- **Chat model:** We will use `gpt-5-nano`, which has a good balance of low cost and reasonable intelligence. You can learn more about this model at the following link: <https://platform.openai.com/docs/models/gpt-5-nano>.

Let's get started:

1. Start your preferred web browser and navigate to the OpenAI signup page at the following link: <https://platform.openai.com/signup>.
2. Once you have signed up and logged in, you will see the OpenAI developer platform home page, as shown in *Figure 9.1*:

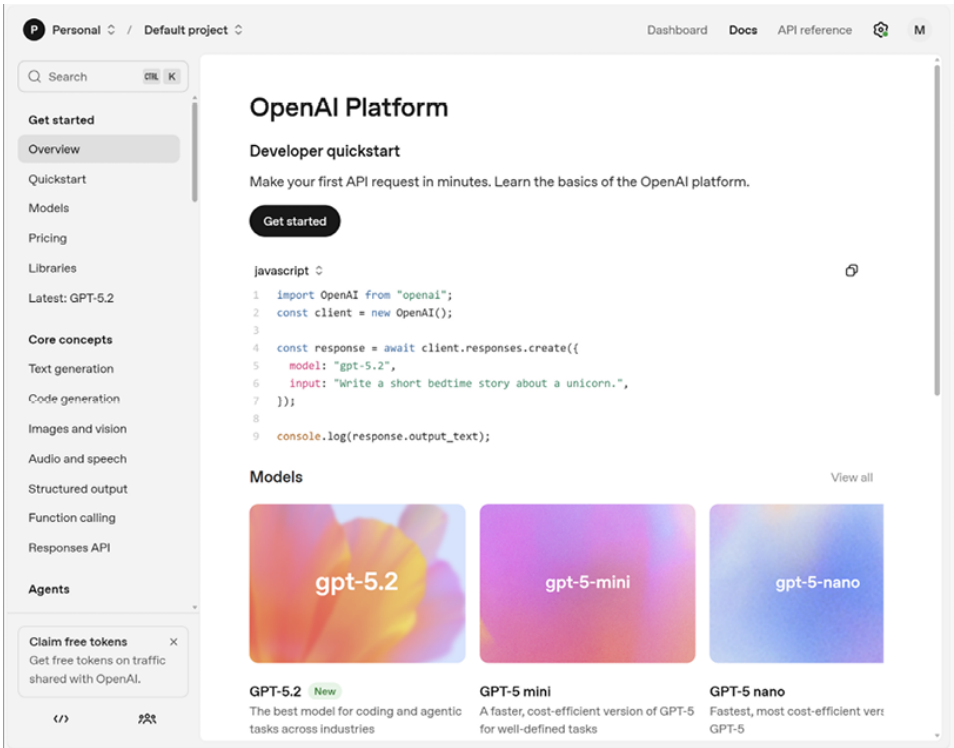


Figure 9.1: OpenAI developer platform home page

1. In the top navigation menu, click the gear icon for Settings (1). Then, in the left navigation menu, navigate to **API keys** (2), and click the **+ Create new secret key** button (3), as shown in Figure 9.2:

Alternatively, you can navigate directly to the page using the following link:

<https://platform.openai.com/settings/organization/api-keys>.

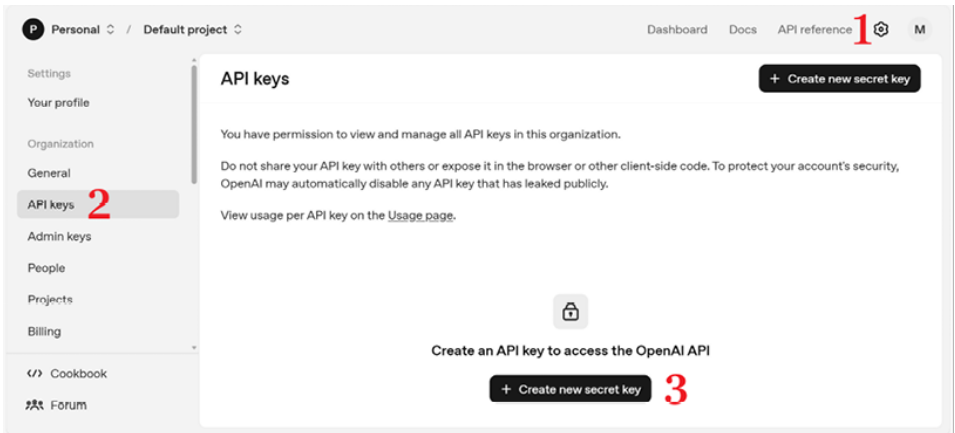


Figure 9.2: The API keys page for your OpenAI account

1. Enter a name like `apps-services-net10`, select a project like **Default Project**, set **Permissions** to **All**, and then click **Create secret key**.
2. When you see the **Save your key** dialog box, copy your key to the clipboard and paste it somewhere safe and accessible.

Good practice: Save this secret key somewhere safe and accessible because you will not be able to view it again through your OpenAI account. If you lose the key, then you will have to generate a new one.

1. In the left navigation menu, navigate to **Billing**, click **Payment methods**, and then add a new payment method, like a Visa card, or navigate to <https://platform.openai.com/settings/organization/billing/payment-methods>.
2. On the **Billing** page, on the **Overview** tab, click the **Add to credit balance** button, and add some credit. You are likely to only spend a few cents, and you can always top it up later if needed, so I recommend putting in the minimum. I added \$5, as shown in Figure 9.3 (which, with

UK tax, cost me \$6):

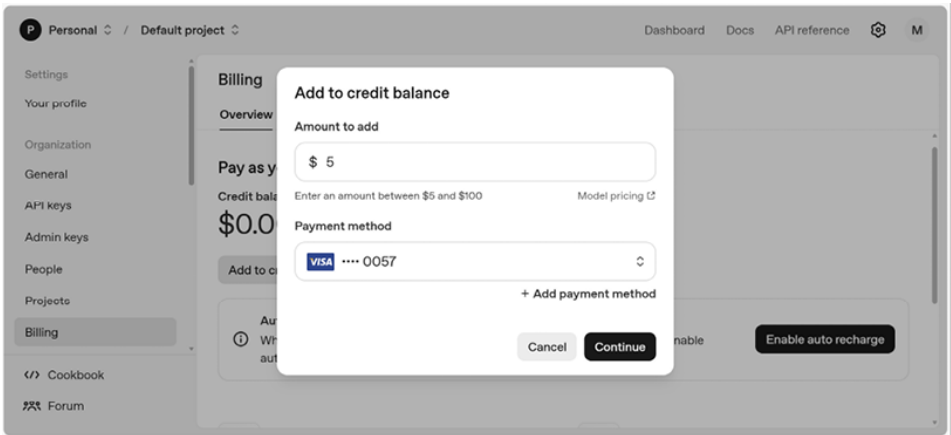


Figure 9.3: Adding credit to your account

1. In the top navigation menu, navigate to **Dashboard**. Then, to test your credit, click the + button, make sure that the **gpt-5-nano** model is selected, enter the following user message: What top three skills should a .NET developer know?, and then note the response, as shown in *Figure 9.4*:

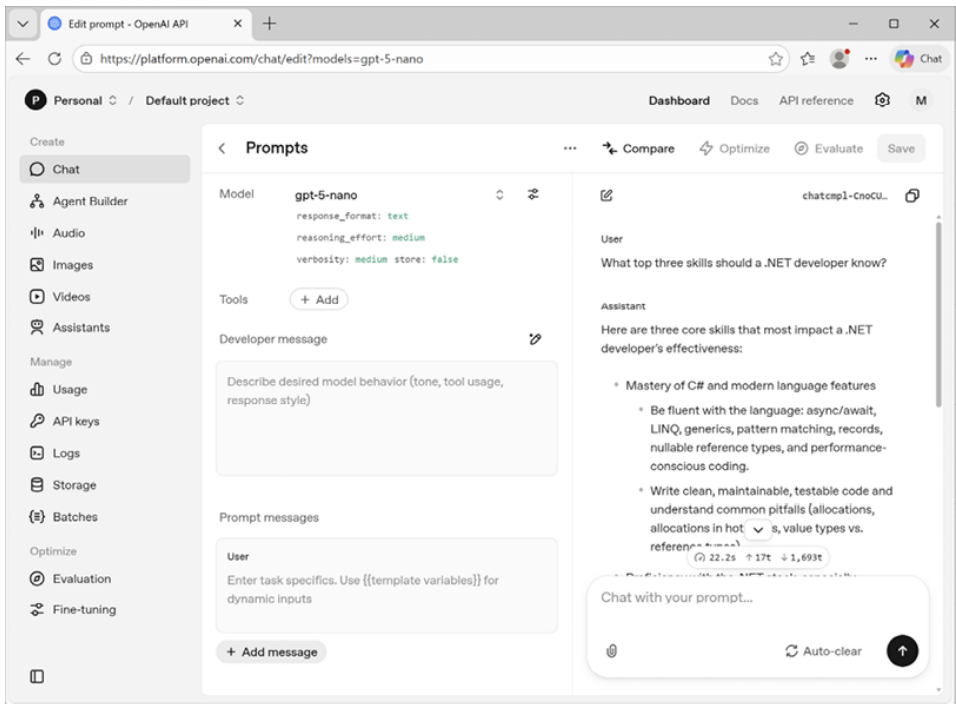


Figure 9.4: Testing your credit using the Dashboard

Warning! If you get an error about not having enough credit, then wait a few minutes for their system to recognize that you've added credit recently. One of the reasons I am getting you to try the Playground now is to make sure that your credit has been recognized before we try calling the API programmatically from .NET code.

1. In the left navigation menu, click **Usage**, or navigate to <https://platform.openai.com/usage>. You can see that the API calls used in this section should only cost a few cents, as shown in Figure 9.5:

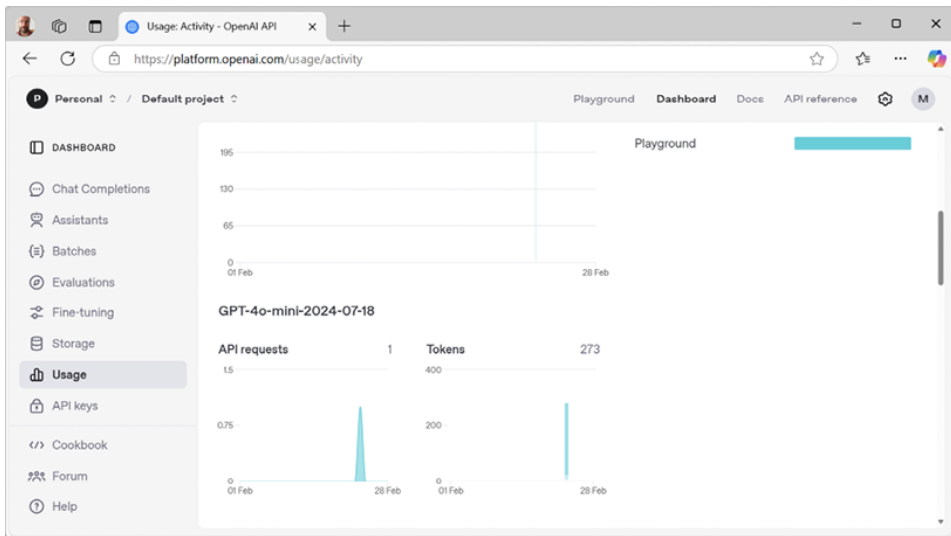


Figure 9.5: Charting API call usage

1. Optionally, close the browser.

You’ve signed up for OpenAI’s API through their administration website. Now, let’s learn how we can integrate this LLM model into a .NET project.

Using Agent Framework with an OpenAI model

The AI experts who build LLMs tend to use Linux as their operating system and Python for programming. Tutorials and documentation tend to assume developers will use those platforms, too. Using another programming language or framework that’s not natively designed for .NET, and dealing with cross-language or cross-framework issues can introduce complexity and reduce the maintainability of your project.

Microsoft Agent Framework is effectively the “next generation” consolidation that takes ideas from Semantic Kernel and another project called

AutoGen and merges them into one unified framework. You can learn more about this transition at the following link: <https://learn.microsoft.com/en-us/agent-framework/overview/agent-framework-overview>.

Introducing Microsoft Agent Framework

Microsoft Agent Framework can leverage the asynchronous programming model of .NET to handle input/output (I/O)-bound tasks more efficiently, which is crucial when integrating network-bound services like an LLM API. This results in a non-blocking I/O operation, which is essential for maintaining the responsiveness of your application.

Agent Framework is designed with the .NET ecosystem in mind, providing integration with C# and other .NET languages. This means it's naturally easier to work with from a .NET developer's perspective, offering libraries or packages that are idiomatic to C# developers.

Using Agent Framework allows for more customizable integration into your .NET project, offering the ability to fine-tune how the LLM interacts with your application. Whether it's adjusting the request pipeline, modifying how responses are processed, or integrating with other .NET services, you have more control over the integration. This flexibility also means you can scale your application more effectively, adjusting the use of the LLM as your project grows.

Given the popularity of .NET, integrating through a method like Agent Framework may provide better community support and resources tailored to .NET developers. This includes documentation, forums, and third-party tools, which can be invaluable for troubleshooting and enhancing your application.

Although you can start with any .NET project and add the Agent Framework packages to it, and then write code yourself to call the LLM, the easiest way to

start is with a project template that implements core functionality using Agent Framework.

Understanding the AI Chat Web App project template

To install the AI Chat Web App project template, at the command prompt or terminal, enter the following command:

```
dotnet new install  
Microsoft.Extensions.AI.Templates
```

The project template will now be available in Visual Studio and other code editors, or at the command line or terminal.

The project template is based on a Blazor Interactive Server web app preconfigured with common AI and data services. The app template handles the following concerns for you:

- Includes essential `Microsoft.Extensions.AI` packages and other dependencies in the `.csproj` file to help you get started working with AI.
- Creates various AI services and registers them for dependency injection in the `Program.cs` file:
 - An `IChatClient` service to chat back and forth with the generative AI model.
 - An `IEmbeddingGenerator` service that's used to generate embeddings, which are essential for vector search functionality.
 - A `JsonVectorStore` to act as an in-memory vector store.
- Registers a SQLite database context service to handle ingesting documents. The app is preconfigured to ingest whatever documents you

add to the `wwwroot/Data` folder of the project, including the provided example files.

- Provides a complete chat UI using Blazor components. The UI handles rich formatting for the AI responses and provides features such as citations for response data.

Warning! The AI Chat Web App project template is still in preview in January 2026. This means that it is likely to change during the lifetime of this book.

Now, we are ready to build a .NET project to call the OpenAI API service using your secret key.

Building the chat app

Let's see the AI Chat Web App project template in action:

1. Use your preferred code editor to add a new **AI Chat Web App** / `aichatweb` project named `ChatApp` to the `ModernApps` solution using the following options:
 - **Framework:** **.NET 10.0 (Long Term Support)**
 - **AI service provider:** **OpenAI Platform**
 - **Vector store:** **Local on-disk (for prototyping)**
 - **Use keyless authentication for Azure services:** **Selected**
 - **Use Aspire orchestration:** **Cleared**

The last two options are not relevant for our project since we are not using Azure or Aspire.

1. In `ChatApp.csproj`, delete any package versions because we are

using CPM, note the packages referenced, and note that user secrets are enabled with a GUID value, as shown highlighted in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup> <TargetFramework>net10.0</
  TargetFramework> <ImplicitUsings>enable</
  ImplicitUsings> <Nullable>enable</
  Nullable>
  <UserSecretsId>73f065dd-9df6-4be6-
  a12c-3b42bc25c31f</UserSecretsId> </
  PropertyGroup> <ItemGroup>
    <PackageReference
      Include="Microsoft.Extensions.AI.OpenAI" /
    > <PackageReference
      Include="Microsoft.Extensions.AI" />
    <PackageReference
      Include="Microsoft.Extensions.DataIngestion"
    /> <PackageReference
      Include="Microsoft.Extensions.DataIngestion.Marko
    /> <PackageReference Include="PdfPig" />
    <PackageReference
      Include="Microsoft.ML.Tokenizers.Data.Cl100kBase"
    /> <PackageReference
      Include="Microsoft.ML.Tokenizers.Data.O200kBase"
    /> <PackageReference
      Include="Microsoft.SemanticKernel.Connectors.Sqlite
    /> </ItemGroup> </Project>
```

2. Build the ChatApp project to restore packages.
3. If you are using Visual Studio, right-click the ChatApp project, select **Manage User Secrets**, and then add an entry for your OpenAI key, as shown in the following JSON:

```
{ "OpenAi:Key": "<your-OpenAI-key>" }
```

4. If you are using any other code editor, you can use the `dotnet` CLI, as shown in the following command:

```
dotnet user-secrets set OpenAi:Key <your-OpenAI-key>
```

This provides the key for your OpenAI service that the app will use to connect and authenticate. By default, the app template searches for this value in the project's local `.NET` user secrets.

1. In the `wwwroot\Data` folder, delete the two existing files, and replace them with the following two files in the GitHub repository for this book:
- https://github.com/markjprice/apps-services-net10/tree/main/code/ModernApps/ChatApp/wwwroot/Data/B31467_Appendix_A.pdf
 - <https://github.com/markjprice/apps-services-net10/tree/main/code/ModernApps/ChatApp/wwwroot/Data/getting-help.md>
4. In the `Components\Pages\Chat` folder, in `Chat.razor`, change the two linked files to reference the two new files, as shown highlighted in the following markup:

```
<ChatMessageList Messages="@messages"  
InProgressMessage="@currentResponseMessage">  
<NoMessagesContent> <div>To get started,
```

try asking about these example documents.
You can replace these with your own data
and replace this message.</div>
<ChatCitation File="B31467_Appendix_A.pdf"
> <ChatCitation File="getting-help.md" >
</NoMessagesContent> </ChatMessageList>

When you start the app, the data ingestion code located in /Services/Ingestion/DataIngestor.cs will compare the contents of the Data folder. It will remove old files from the configured vector store and add new ones.

Warning! If you have too many files, or too large files, you may run into quota and rate limits. When you hit a limit, you may see an error message or experience long delays in the startup of the application.

1. In Program.cs, change the OpenAI model name, as shown in the following code:

```
var chatClient =  
openAIClient.GetResponseClient( "gpt-5-  
nano").AsIChatClient();
```

At the time of writing in January 2026, the model in the project template was still gpt-4o-mini, but this is likely to change over time. Other model names include: gpt-5.2, gpt-5.2-pro, and

gpt-5.2-chat-latest.

Trying out the chat app

Now we can start the chat app and try out its functionality:

1. Start the ChatApp project without debugging.
2. Enter the following question: What is in Appendix A?. Then, note the response (which, of course, could be different for you since there is inherent randomness involved), as shown in *Figure 9.6*:

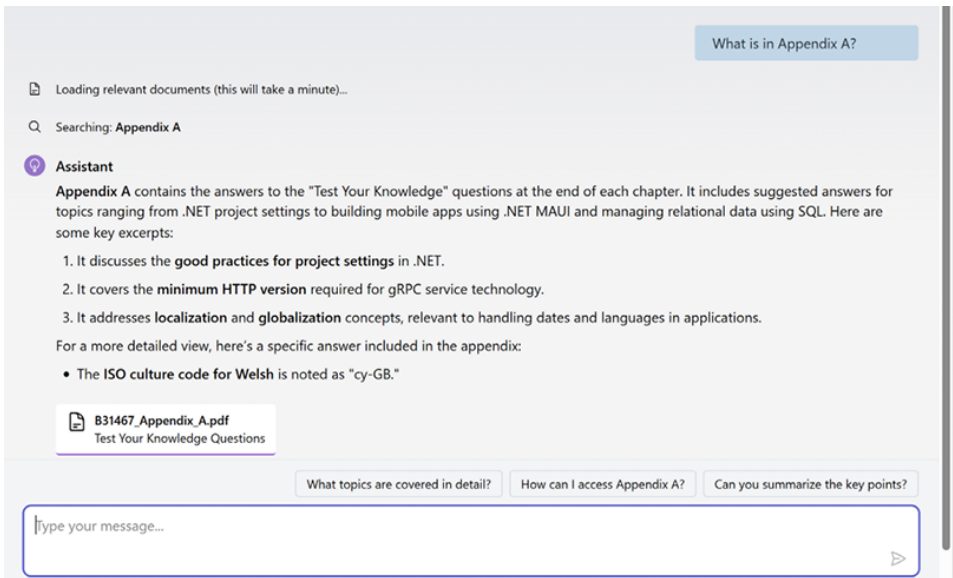


Figure 9.6: Answering a question about Appendix A

1. Enter another question: What is some advice for asking questions on Discord? Then, note the response, which should be a numbered list based on the contents of the getting-help.md file.
2. Enter another question: Please write a short biography of any Mark Price that you are aware of. Then, note the response, as shown in the following output:

Mark Price (born February 15, 1967) is a former professional basketball player best known for his time in the NBA, particularly with the Cleveland Cavaliers. Born in Bartlesville, Oklahoma, Price excelled in basketball from a young age. He attended Enid High School and later played college basketball at Georgia Tech, where he gained recognition for his skills as a point guard. Price was selected by the Cleveland Cavaliers in the 1986 NBA Draft. Throughout his career, he became known for his exceptional shooting ability, particularly from three-point range, and his impressive passing skills. He was a two-time NBA All-Star (1989, 1990) and won the NBA's free throw percentage title multiple times. ...

The most famous Mark Price on the internet is a basketball player.

1. Close the browser and shut down the web app.

Practical AI chatbot applications usually need to work with data about an organization that is in a relational database or another data store.

Now, let's learn about functions that can give our chatbot more information to help it help us.

Understanding functions

In the context of OpenAI models, functions refer to specific, callable features or operations that these models can execute. These functions extend the capabilities of language models beyond text generation, enabling them to perform more

structured and task-specific actions.

Functions in OpenAI models are predefined operations or tasks that the model can execute when invoked. These functions allow the model to perform specialized actions based on the input it receives, enhancing its utility and adaptability for various applications.

Functions can vary widely depending on their intended use. Here are some common types:

- **Text manipulation:** Operations like summarizing text, translating between languages, or extracting specific information from a given text.
- **Data analysis:** Performing calculations, generating charts, analyzing datasets, or extracting patterns from data.
- **Code assistance:** Functions that help with writing, debugging, and understanding code in various programming languages.
- **Interaction with APIs:** Functions designed to interact with external systems, databases, or APIs to fetch or send information. This is the type that we will add to extend the OpenAI model with custom information.

Functions typically follow a structured process:

- **Invocation:** A function is called or triggered based on specific keywords or instructions within the user's input. This can be explicit (directly calling a function) or implicit (the model recognizing a task that maps to a function).
- **Processing:** Once invoked, the function processes the input according to its defined logic or algorithm. This may involve parsing text, performing computations, querying databases, or interacting with APIs.
- **Output generation:** After processing, the function generates an output that is returned to the model so it can add and process it before showing it to the user. This output can be a text response, a data structure, or any other relevant format.

Benefits of functions include:

- Streamlining complex tasks, providing quick and accurate responses for specific queries.
- Enabling the model to handle specialized tasks that require more than just general language understanding.
- Integrating with external systems, databases, and APIs, making the model a versatile tool for a wide range of applications.

Functions are a powerful extension of OpenAI model capabilities, enabling them to perform specialized tasks and interact with external systems efficiently.

Now, let's add a function to our kernel to give our chatbot more information about me and the Northwind database.

Adding functions

You can configure multiple functions and enable the OpenAI service API to decide for itself when to use them, based on textual descriptions and the user prompt.

Let's explore:

1. Modify `ChatApp.csproj` to add a reference to the Northwind data context class library project, as shown in the following markup:

```
<ItemGroup> <ProjectReference Include= "..  
  \Northwind.DataContext  
  \Northwind.DataContext.csproj" /> </  
ItemGroup>
```

2. Build the `ChatApp` project to compile the referenced project and copy its assembly to the local project's `bin` folder.
3. In the `Components\Pages\Chat` folder, in `Chat.razor`, import namespaces for working with EF Core, your data context, and JSON serialization, as shown highlighted in the following code:


```
@page "/" @using System.ComponentModel
@using Microsoft.EntityFrameworkCore; @*
To use Include. *@ @using
Northwind.EntityModels; @* To use
NorthwindContext. *@ @using
System.Text.Json; @* To use
JsonSerializer. *@
```

4. In `Chat.razor`, in the Razor code block, define some methods to get information about me and a subset of fields for products in each category of the Northwind database, as shown in the following code:

The complete file can be found at the following link: <https://github.com/markjprice/apps-services-net10/tree/main/code/ModernApps/ChatApp/Components/Pages/Chat/Chat.razor>.

```
@code { private string GetAuthorBiography() {
return "" Mark J Price is a former Microsoft
Certified Trainer (MCT) and current Microsoft
Specialist: Programming in C# and Architecting
Microsoft Azure Solutions, with more than 20
years' of educational and programming
experience. Since 1993 Mark has passed more
than 80 Microsoft programming exams and
specializes in preparing others to pass them
too. His students range from professionals
with decades of experience to 16-year-old
apprentices with none. Mark successfully
guides all of them by combining educational
```

skills with real-world experience consulting and developing systems for enterprises worldwide. Between 2001 and 2003 Mark was employed full-time to write official courseware for Microsoft in Redmond, USA. Mark's team wrote the first training courses for C# while it was still an early alpha version. While with Microsoft he taught "train-the-trainer" classes to get other MCTs up-to-speed on C# and .NET. Between 2016 and 2022, Mark created and delivered training courses for Optimizely's Digital Experience Platform, the best .NET CMS for Digital Marketing and E-commerce. In 2010 Mark studied for a Post-Graduate Certificate in Education (PGCE). He taught GCSE and A-Level mathematics in two London secondary schools. Mark holds a Computer Science BSc. Hons. Degree from the University of Bristol, UK. Mark J Price has authored the following books: 1. C# 6 and .NET Core 1.0 - Modern Cross-Platform Development, 1st Edition, 2016 2. C# 7 and .NET Core - Modern Cross-Platform Development, 2nd Edition, 2017 3. C# 7.1 and .NET Core 2.0 - Modern Cross-Platform Development, 3rd Edition, 2017 4. C# 8.0 and .NET Core 3.0 - Modern Cross-Platform Development, 4th Edition, 2019 5. C# 9 and .NET 5 - Modern Cross-Platform Development, 5th Edition, 2020 6. C# 10 and .NET 6 - Modern Cross-Platform Development, 6th Edition, 2021 7. C# 11 and .NET 7 - Modern Cross-Platform Development Fundamentals, 7th Edition, 2022 8. Apps and Services with .NET 7, 1st Edition, 2022 9. C# 12 and .NET 8 - Modern Cross-Platform

Development Fundamentals, 8th Edition, 2023
 10. Apps and Services with .NET 8, 2nd Edition, 2023
 11. Tools and Skills for .NET 8, 1st Edition, 2024
 12. C# 13 and .NET 9 - Modern Cross-Platform Development Fundamentals, 9th Edition, 2024
 13. Real-World Web Development with .NET 9, 1st Edition, 2024
 14. C# 14 and .NET 10 - Modern Cross-Platform Development Fundamentals, 10th Edition, 2025
 15. Real-World Web Development with .NET 10, 2nd Edition, 2025
 16. Apps and Services with .NET 10, 3rd Edition, 2026
 17. Tools and Skills for .NET 10, 2nd Edition, 2026

```
""; }
// It is more efficient to cache this than
create it every time. private
JsonSerializerOptions jsonSerializerOptions =
new() { WriteIndented = true }; private string
GetProductsInCategory(string categoryName) {
using NorthwindContext db = new(); var
products = db.Products .Include(p =>
p.Category) .Where(p =>
p.Category!.CategoryName == categoryName)
.Select(p => new { p.ProductId, p.ProductName,
p.Category!.CategoryName, p.UnitPrice,
p.UnitsInStock, p.UnitsOnOrder, p.Discontinued
}) .ToArray(); // Convert the array of
products to a JSON string. string json =
JsonSerializer.Serialize( products,
jsonSerializerOptions); return json; } ... }
```

I've chosen a subset of fields for products that will provide enough data to the model to be able to answer some interesting questions directly about products in

categories. I have excluded related tables like orders and suppliers to minimize what gets serialized. This means the model will not be able to answer questions like, “Who supplies Chai?” (because the model does not have information about suppliers) or “How many customers order Chang?” (because the model does not have information about orders).

1. In `Chat.razor`, in the `OnInitialized` method, add statements to define two functions from our two methods, as shown highlighted in the following code:

```
protected override void OnInitialized() {  
    statefulMessageCount = 0;  
    messages.Add(new(ChatRole.System,  
        SystemPrompt)); chatOptions.Tools = [  
        AIFunctionFactory.Create(LoadDocumentsAsync),  
        AIFunctionFactory.Create(SearchAsync),  
        AIFunctionFactory.Create( method:  
            GetAuthorBiography, name:  
            nameof(GetAuthorBiography), description:  
            "Gets the biography of the author Mark  
            Price."), AIFunctionFactory.Create(  
            method: GetProductsInCategory, name:  
            nameof(GetProductsInCategory),  
            description: "Get the products in a  
            category from the Northwind database.") ];  
    }  
}
```

2. In `Chat.razor`, in the `SystemPrompt` string constant, add statements to tell the LLM when to use the two functions, as shown

highlighted in the following code:

```
private const string SystemPrompt = @" You
are an assistant who answers questions
about information you retrieve. Do not
answer questions about anything else. Use
only simple markdown to format your
responses. Use the GetAuthorBiography tool
to get the biography of Mark Price. Use
the GetProductsInCategory tool to get
products from the Northwind database. Use
the LoadDocuments tool to prepare for
searches before answering any questions.
...
```

Trying out the chat app with custom functions

Now we can try out the chat app in ways that will call the two functions.

1. Start the ChatApp project without debugging.
2. Enter the following instruction: Please write a short biography of any Mark Price that you are aware of. Then, note the response, as shown in the following output:

```
Mark J Price is a veteran Microsoft
trainer and author. A former Microsoft
Certified Trainer and current Microsoft
Specialist in Programming in C# and
Architecting Microsoft Azure Solutions, he
has over 20 years of experience in
teaching and software development. He
contributed to official Microsoft
```

courseware, helped create early C# training, and later taught "train-the-trainer" programs. He holds a BSc in Computer Science from the University of Bristol and a PGCE, and has written numerous books on C# and .NET across editions from 2016 onward.

3. Enter the following question: When did the author write a book about C# 10?. Then, note the response, as shown in the following output:

The author, Mark J Price, wrote a book about C# 10 titled "C# 10 and .NET 6 - Modern Cross-Platform Development" in 2021.

4. Enter the following question: How many books has Mark written?. Then, note the response, as shown in the following output:

Mark J Price has written a total of 17 books.

5. Enter the following question: Where did the author learn C#?. Then, note the response, as shown in the following output:

Mark J Price learned C# while he was employed full-time at Microsoft between 2001 and 2003, where he was part of a team that wrote the first training courses for C# during its early alpha version. He also taught "train-the-trainer" classes to help

other Microsoft Certified Trainers (MCTs) become proficient in C# and .NET. His extensive experience and focus on programming education contribute to his expertise in C#.

6. Enter the following question: What is the most expensive seafood sold by Northwind?. Then, note the response, as shown in the following output:

The most expensive seafood in Northwind is Carnarvon Tigers, priced at 62.50 per unit.

7. Enter the following question: What products are in the seafood category?. Then, note the response, as shown in the following output:

Here are the products in the Seafood category: 1. Ikura 2. Konbu 3. Carnarvon Tigers 4. Nord-Ost Matjeshering 5. Inlagd Sill 6. Gravad lax 7. Boston Crab Meat 8. Jack's New England Clam Chowder 9. Rogede sild 10. Spegesild 11. Escargots de Bourgogne 12. Röd Kaviar

8. Enter the following request: Please include the prices. Then, note the response, as shown in the following output:

Here are the products in the Seafood category along with their prices: 1. Ikura - \$31.00 2. Konbu - \$6.00 3. Carnarvon

```
Tigers - $62.50 4. Nord-Ost Matjeshering -  
$25.89 5. Inlagd Sill - $19.00 6. Gravad  
lax - $26.00 7. Boston Crab Meat - $18.40  
8. Jack's New England Clam Chowder - $9.65  
9. Rogede sild - $9.50 10. Spegesild -  
$12.00 11. Escargots de Bourgogne - $13.25  
12. Röd Kaviar - $15.00
```

9. Enter the following request: Please generate the source code HTML for a table of the products in the condiments category. Please include the price and number in stock.
10. Then, note the response, as shown in the following partial output:

```
<table> <thead> <tr> <th>Product Name</th>  
<th>Unit Price</th> <th>Units In Stock</  
th> </tr> </thead> <tbody> <tr>  
<td>Aniseed Syrup</td> <td>$10.00</td>  
<td>13</td> </tr> ... <tr> <td>Original  
Frankfurter grüne Soße</td> <td>$13.00</  
td> <td>32</td> </tr> </tbody> </table>
```

11. Close the browser and shut down the web server.

You've now seen how to build a simple chat app that integrates a cloud-based LLM with custom functions.

Now, let's see an alternative to writing custom functions: integrating AI models with a custom MCP server.

Building an MCP server

During 2025, you probably heard the term MCP a lot in relation to LLMs.

Understanding the Model Context Protocol

At its core, an MCP server is part of something called the **Model Context Protocol (MCP)**. MCP is an open standard for letting AI models like large language models (LLMs) connect with real-world data, tools, and services in a reliable, predictable way. The protocol was released in late 2024 by Anthropic and is now being adopted across major AI ecosystems.

An MCP server is a program or service that:

- Exposes specific capabilities, data sources, or tools via a standard interface that AI models can use.
- Talks a protocol the AI understands instead of relying on bespoke custom integration code.

Typical examples could be servers that let an AI read documents, query a database, access your calendar, or trigger a deployment pipeline.

Before MCP, you had to write bespoke code every time you wanted an AI model to talk to a new system (database, CRM, Git, NuGet, and so on). MCP lets you wrap those systems in a standardized interface that all MCP-aware clients can “plug into” without rewriting everything.

There are three main pieces:

1. **MCP host:** The application using the LLM (like a chatbot or IDE).
2. **MCP client:** A library inside the host that manages the MCP connection.
3. **MCP server:** The service exposing tools or data using the MCP standard.

When the AI needs something external, it asks the MCP client. The client forwards that to the MCP server. The server executes the request, like running a database query, and returns a structured result that the model can understand.

Benefits of MCP

The benefits stack up mainly around making AI more useful in real systems:

1. MCP servers open access to real, live systems like databases, APIs, files, calendars, and more, so outputs aren't limited to what the model memorized during training.
2. Instead of building unique integrations for each model and each system, MCP lets teams wrap their systems once and reuse that connector everywhere. That cuts development effort and saves time.
3. MCP servers let models get up-to-date data directly from the source rather than relying on cached or embedded datasets. That's way better for tasks that need current context.
4. Since capabilities are exposed as standard interfaces, systems can mix and match connectors without tight coupling. That makes apps easier to scale and evolve.
5. By isolating data access behind an MCP server interface, you can enforce consistent, auditable access controls rather than letting AI models reach everywhere in an ad-hoc way.
6. MCP servers drive coordination across different tools and services for complex agents that might read docs, send emails, post to Slack, and update a database in one workflow.

In simple terms, an MCP server is the bridge that lets AI systems access real tools and real data through a single, reusable, open standard. Without it, you'd have to write custom code for each model and each tool. With it, you wrap each system once, and then any MCP-aware AI client can use it.

If you're building AI apps that need to go beyond plain text responses and actually integrate with the world around them, MCP servers are rapidly becoming the go-to approach.

So, how do we build an MCP using .NET?

Using .NET to build an MCP server

MCP was originally designed by Anthropic, so it makes sense that Microsoft partnered with Anthropic to create an official C# SDK for Model Context

Protocol, and you can read about this at the following link: <https://developer.microsoft.com/blog/microsoft-partners-with-anthropic-to-create-official-c-sdk-for-model-context-protocol>.

Warning! Like many technologies related to AI, the MCP C# SDK is in preview, and APIs may change.

The MCP C# SDK is distributed as a NuGet package named `ModelContextProtocol` that you can integrate into any .NET project. This package gives access to APIs to create clients that connect to MCP servers, the creation of MCP servers, and AI helper libraries to integrate with LLMs through `Microsoft.Extensions.AI` and Agent Framework.

Let's build a simple console app:

1. Use your preferred code editor to add a new **Console App** / `console` project named `NorthwindMcp` to the `ModernApps` solution.
2. In the `NorthwindMcp.csproj` project file, add references to packages for the MCP C# SDK, hosting, and Spectre Console, a project reference to your Northwind data context, and statically and globally import the `System.Console` type, as shown in the following markup:

```
<ItemGroup> <PackageReference
Include="ModelContextProtocol" />
<PackageReference
Include="Microsoft.Extensions.Hosting" />
<PackageReference
Include="Spectre.Console" /> </ItemGroup>
<ItemGroup> <ProjectReference Include= "..
\Northwind.DataContext
\Northwind.DataContext.csproj" /> </
ItemGroup> <ItemGroup> <Using
```

```
Include="System.Console" Static="true" />
</ItemGroup>
```

3. In the NorthwindMcp project, add a new folder named Tools.
4. In the Tools folder, add a new class named HelloMcpTool.cs.
5. In HelloMcpTool.cs, add a statement to define a class, as shown in the following code:

```
using ModelContextProtocol.Server; // To
use [McpServerToolType] and
[McpServerTool]. using
System.ComponentModel; // To use
[Description]. [McpServerToolType] public
static class HelloMcpTool {
[McpServerTool] [Description("Responds
with a greeting back to the client.")]
public static string HelloMcp(string name)
=> $"Hello, {name}"; }
```

6. In the Tools folder, add a new class named HelloMcpTool.cs.
7. In HelloMcpTool.cs, add a statement to define a class, as shown in the following code:

```
using Microsoft.EntityFrameworkCore; // To
use .Include() extension method. using
ModelContextProtocol.Server; // To use
[McpServerToolType] and [McpServerTool].
using Northwind.EntityModels; // To use
NorthwindContext. using
System.ComponentModel; // To use
[Description]. using System.Text.Json; //
To use JsonSerializer. [McpServerToolType]
public static class NorthwindTool { // It
```

```

is more efficient to cache this than
create it every time. private static
JsonSerializerOptions
jsonSerializerOptions = new() {
WriteIndented = true }; [McpServerTool]
[Description("Responds with a greeting
back to the client.")] public static
string Products(string? categoryName =
null) { using NorthwindContext db = new();
IEnumerable<Product> products =
db.Products .Include(p => p.Category); if
(!string.IsNullOrEmpty(categoryName)) {
products = products.Where(p =>
p.Category!.CategoryName == categoryName);
} var filteredProducts = products.Select(p
=> new { p.ProductId, p.ProductName,
p.Category!.CategoryName, p.UnitPrice,
p.UnitsInStock, p.UnitsOnOrder,
p.Discontinued }) .ToArray(); // Convert
the array of products to a JSON string.
string json = JsonSerializer.Serialize(
filteredProducts, jsonSerializerOptions);
return json; } }

```

8. In Program.cs, delete any existing statements, and then add statements to x, as shown in the following code:

```

using
Microsoft.Extensions.DependencyInjection;
// To use AddMcpServer. using
Microsoft.Extensions.Hosting; // To use
Host. using Microsoft.Extensions.Logging;
// To use AddConsole. var builder =
Host.CreateApplicationBuilder(args);

```

```
builder.Logging.AddConsole(consoleLogOptions
=> { // Configure all logs to go to stderr
consoleLogOptions.LogToStandardErrorThreshold
= LogLevel.Trace; }); builder.Services
.AddMcpServer()
.WithStdioServerTransport()
.WithToolsFromAssembly(); await
builder.Build().RunAsync();
```

The `WithToolsFromAssembly` method tells the MCP server to scan the assembly for classes with the `McpServerToolType` attribute and register all methods with the `McpServerTool` attribute. The `Description` attribute helps the client determine which tool to call.

1. In the `ModernApps` folder, create a new file named `.mcp.json`.
2. In `.mcp.json`, define an MCP server that starts up the `NorthwindMcp` project, as shown in the following markup:

```
{ "inputs": [], "servers": {
  "NorthwindMcp": { "type": "stdio",
    "command": "dotnet", "args": [ "run", "--
project", "C:\\apps-services-net10\\
\\ModernApps\\NorthwindMcp\\
\\NorthwindMcp.csproj" ] } } }
```

Warning! Make sure the path to your project is correct.

Trying out the MCP server in your code editor

Now we can try out the MCP server in Visual Studio, VS Code, or any app that can connect to MCP servers:

1. In Visual Studio, in **GitHub Copilot Chat**, click the **Select tools** button, clear any other tools, and select the two **NorthwindMcp** tools, as shown in *Figure 9.7*:

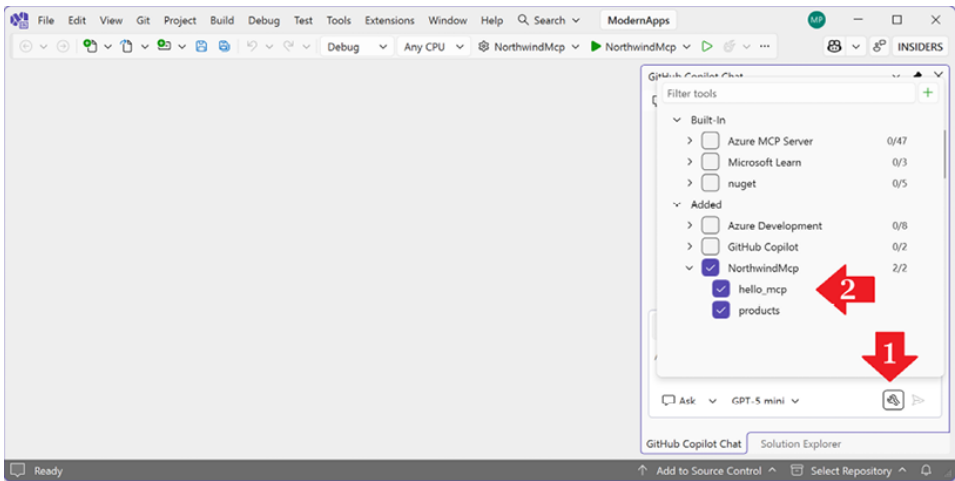


Figure 9.7: Selecting your NorthwindMcp tools for use by GitHub Copilot Chat

1. In the chat box, enter the prompt, What products are in the seafood category?, and note that GitHub Copilot Chat will realize it should run your MCP server tool, and ask you to confirm that it has permission to do so, as shown in *Figure 9.8*:

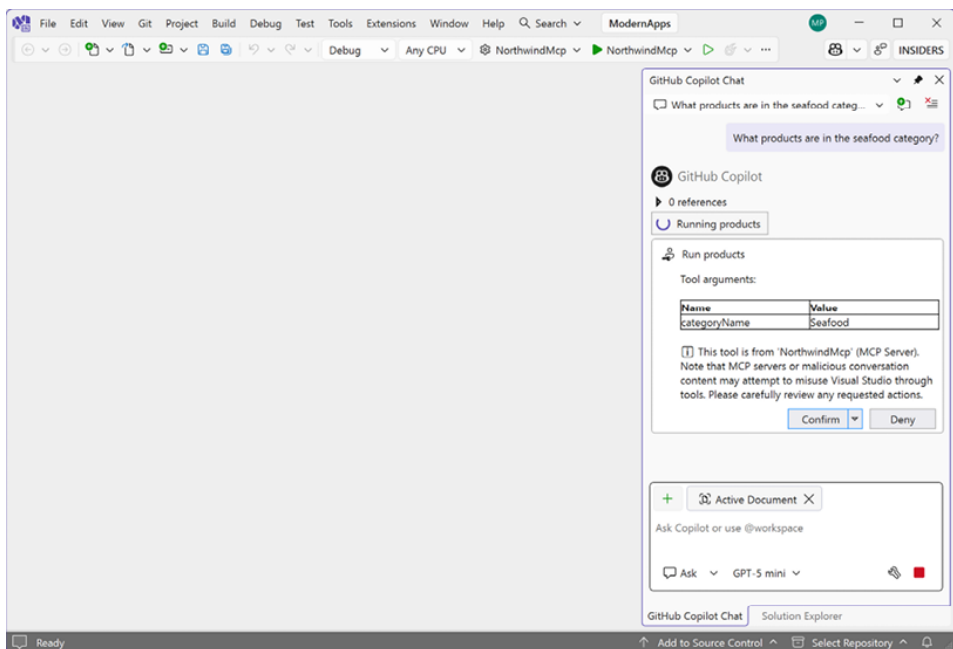


Figure 9.8: You must confirm that you want to run the products tool

1. Click the **Confirm** button, and note a table of products in the seafood category is returned, as shown in Figure 9.9:

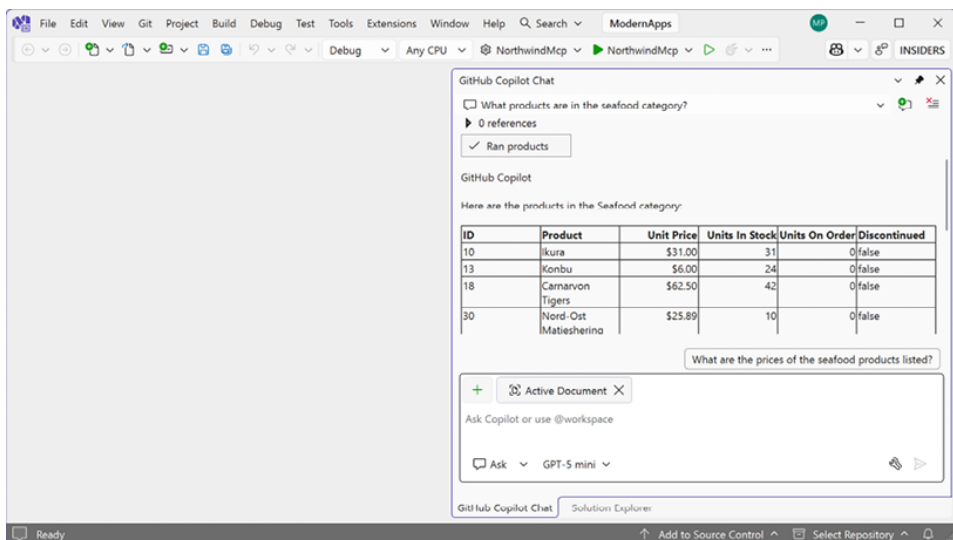


Figure 9.9: A table of products in the seafood category

If you have any issues with the MCP server, it could be a misconfiguration, or the MCP server might need to be restarted. You can do this by clicking the three-dots menu to the right of any MCP server, clicking **Edit .mcp.json**, and then making sure the MCP server is running properly, as shown in Figure 9.10:

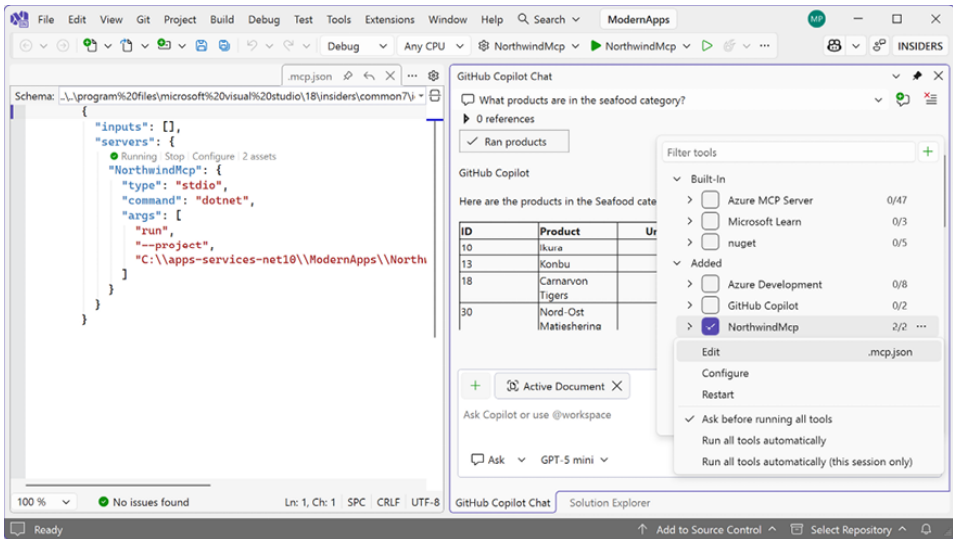


Figure 9.10: Ensuring the MCP server is running properly

There is a lot more that you can do with MCP, but for now, let's move on to running LLMs locally instead of in the cloud.

Running local LLMs

In this section, we will look at how you can download and run LLMs on your local computer, including tools and platforms like Hugging Face, Ollama, and LM Studio.

Hugging Face

Hugging Face is a popular open-source platform for building and sharing state-of-the-art models in natural language processing. It is known for creating tools,

libraries, and resources that democratize access to advanced machine-learning models. There are several areas where Hugging Face makes contributions to the LLM community:

- **Transformers library:** This provides a wide range of pretrained models for various NLP tasks, such as text generation, translation, summarization, and question-answering.
- **Datasets library:** This provides a vast collection of datasets ready for use in machine learning projects, covering a wide range of tasks and domains.
- **Model hub:** This is a platform where users can upload and share their own models with the community, browse and download models shared by others, and access tools for evaluating and benchmarking model performance.
- **Tokenizers library:** This provides efficient and flexible tokenization methods that are used to prepare text data for transformer models.

You can see what’s trending this week on the Hugging Face home page: <https://huggingface.co/>, and note that DeepSeek’s R1 model has had more than four million downloads, as shown in *Figure 9.11*:

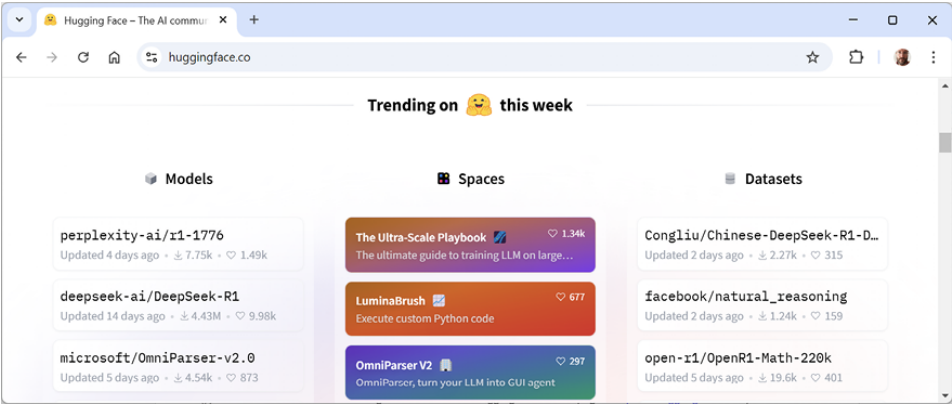


Figure 9.11: The Hugging Face home page

DeepSeek’s R1 model has gained popularity because it offers strong performance, open

accessibility, and cost-effectiveness compared to proprietary alternatives like OpenAI's GPT-5 or Anthropic's Claude. It stands out as a powerful open-weight LLM, which means developers can fine-tune it, deploy it on their own infrastructure, and integrate it into custom applications without API restrictions.

Now, let's look at some tools that can make acquiring and using Hugging Face models locally on your computer as easy as possible.

Ollama

Ollama is a software tool designed for managing LLMs on local devices. This tool allows users to run sophisticated language models, such as **LLaMA (Large Language Model Meta AI)**, on their own machines rather than relying solely on cloud-based solutions.

By running models locally, Ollama ensures that sensitive data remains on the user's device, enhancing privacy and security compared to cloud-based services. Users maintain full ownership and control over their data, which is particularly important for sensitive or proprietary information.

Developers and researchers can use Ollama to experiment with and fine-tune LLMs without the need for cloud resources. Running models locally can be more cost-effective in the long run, as it eliminates the need for continuous cloud service subscriptions. However, local installations require ongoing maintenance and updates, which can be more complex compared to managed cloud services.

To run LLMs locally, users need powerful hardware, typically including high-end CPUs, significant RAM, and often **graphics processing units (GPUs)** to handle the intensive computation. Running models locally reduces latency, which is critical for applications requiring immediate responses. LLMs are

resource-intensive, and not all users may have the necessary hardware to run them efficiently.

Ollama aims to simplify the installation and setup process, providing tools and documentation to help users get started with minimal friction.

Let's download and install Ollama:

1. Start your preferred browser and navigate to: <https://ollama.com/>.
2. On the Ollama home page, click the **Download** button.
3. Select your OS and follow the instructions to install Ollama.

Now let's review Ollama models, its CLI, and the OllamaSharp .NET package for easily integrating Ollama with any .NET project.

Ollama models

Models often come in multiple variations, typically by size and capability. For example:

- `llama` and `llama:70b` vary by size.

Smaller models like `llama` can perform adequately for less complex tasks like basic text generation, simple question answering, and summarization. For applications requiring real-time responses like chatbots, smaller models are generally better due to lower inference latency. More complex tasks such as nuanced understanding, detailed reasoning, and generating longer, coherent text may benefit from larger models like `llama:70b`. But larger models require significantly more computational power

and memory for both training and inference. Also consider that sometimes a smaller, fine-tuned model can outperform a larger, general-purpose model in specific domains.

- The `mistral:instruct` model follows instructions.
- The `mistral:text` model is the base foundation model without any fine-tuning for conversations and is best used for simple text completion.

To quickly download and run an Ollama model in interactive mode, use the following command:

```
ollama run <model>
```

The most common models supported by Ollama include those shown in *Table 9.1*:

[illegible]

The Ollama CLI provides a range of commands to manage and run LLMs locally. While the exact commands and options may vary depending on the version of Ollama and its implementation, the following is a general overview of common CLI commands and their typical usage.

Let's explore what you can do with the Ollama CLI with the Llama 3 model:

1. At the command prompt or terminal, enter the command to check its version, as shown in the following command:

```
ollama --version
```

2. Note the response, as shown in the following output:

```
ollama version is 0.13.5
```

3. At the command prompt or terminal, enter the command to pull down a named model like Meta's Llama3 or Google's Gemma3, as shown in the following command:

```
ollama pull gemma3:1b
```

You can see the Llama3 models at the following link: <https://ollama.com/library/llama3>. You can see all the Gemma3 models at the following link: <https://ollama.com/library/gemma3>.

1. Note the response, as shown in the following output:

```
pulling manifest pulling 6a0746a1ec1a...  
100%  
  
4.7 GB pulling 4fa551d4f938... 100%  
  
12 KB pulling 8ab4849b038c... 100%  
  
254 B pulling 577073ffcc6c... 100%  
  
110 B pulling 3f8eb4da87fa... 100%  
  
485 B verifying sha256 digest writing  
manifest success
```

You can remove a model using the following command: `ollama rm gemma3:1b`.

1. At the command prompt or terminal, enter the command to list the available local models, as shown in the following command:

```
ollama list
```

2. Note the response, as shown in the following output:

```
NAME ID SIZE MODIFIED gemma3:1b  
8648f39daa8f 815 MB 7 seconds ago  
llama3.1:latest 46e0c10c039e 4.9 GB 2  
minutes ago
```

3. At the command prompt or terminal, enter the command to run a named model (which will also download it if not already pulled), as shown in the

following command:

```
ollama run gemma3:1b
```

4. Note the response, as shown in the following output:

```
>>> Send a message (/? for help)
```

5. Enter a prompt, like, `What is .NET?`, and note the response which will use Markdown syntax for formatting.

6. Enter the command to exit: `/bye`

Ollama CLI also has commands to copy models and create new models. You can learn about these and other commands in the documentation at the following link:
<https://github.com/ollama/ollama#cli-reference>.

Ollama only provides client libraries for Python and JavaScript, but a third-party has created a library for .NET developers. We'll look at this next.

OllamaSharp .NET package

OllamaSharp is a .NET binding for the Ollama API, making it easy to interact with Ollama using your .NET. It includes support for all Ollama API endpoints, including chats, embeddings, listing models, and pulling and creating new models.

You can view the OllamaSharp GitHub repository at the following link: <https://github.com/>

Let's see it in action:

1. Use your preferred code editor to add a new **Console App** / console project named `OllamaApp` to the `ModernApps` solution.
2. In the `OllamaApp.csproj` project file, add references to packages for Spectre Console and Ollama, and statically and globally import the `System.Console` type, as shown in the following markup:

```
<ItemGroup> <PackageReference  
  Include="Spectre.Console" />  
<PackageReference Include="OllamaSharp" />  
</ItemGroup> <ItemGroup> <Using  
  Include="System.Console" Static="true" />  
</ItemGroup>
```

3. In `Program.cs`, delete any existing statements and then add statements to:

- Import namespaces to work with Ollama and Spectre Console.
- Create an Ollama client that sends requests to Ollama on localhost using its default port 11434.
- Render a Spectre Console table to show all the models available in your local Ollama installation.
- Set the Ollama client to use the latest Llama 3 model.
- Prompt the user, send the request to Ollama, and output the response.
- Use a stopwatch to measure how long it takes.

```
using OllamaSharp; // To use  
OllamaApiClient. using OllamaSharp.Models;
```

```
// To use Model. using Spectre.Console; //
To use Table. using System.Diagnostics; //
To use Stopwatch. // Default port for the
Ollama API is 11434. string port =
"11434"; Uri uri = new($"http://localhost:
{port}"); OllamaApiClient ollama =
new(uri); Table table = new();
table.AddColumn("Name");
table.AddColumn("Size"); // Get the list
of models. IEnumerable<Model> models =
await ollama.ListLocalModelsAsync();
foreach (Model model in models) {
table.AddRow(model.Name,
model.Size.ToString("N0")); }
AnsiConsole.Write(table); string modelName
= "gemma3:1b"; WriteLine();
WriteLine($"Selected model: {modelName}");
ollama.SelectedModel = modelName;
ConsoleKey key = ConsoleKey.A; while (key
is not ConsoleKey.X) { Write("Please enter
your prompt: "); string? prompt =
ReadLine(); if
(string.IsNullOrEmpty(prompt)) {
WriteLine("Prompt is required. Exiting the
app."); return; } Stopwatch timer =
Stopwatch.StartNew(); await foreach
(GenerateResponseStream? response in
ollama.GenerateAsync(prompt)) {
Write(response?.Response); } timer.Stop();
WriteLine(); WriteLine($"Elapsed time:
{timer.ElapsedMilliseconds:N0} ms");
timer.Restart(); WriteLine();
WriteLine("Press X to exit or any other
key to ask another question."); key =
ReadKey(intercept: true).Key; }
```

10. Start the OllamaApp project without debugging.
11. Enter a prompt like What is .NET in one paragraph? and note the result, as shown in the following output:

```
Please enter your prompt: What is .NET in
one paragraph? .NET (pronounced "dot net")
is a free and open-source framework
developed by Microsoft that provides a
software development platform for building
various types of applications, including
Windows desktop and mobile apps, web
applications, and services. It consists of
a large library of pre-built code
components called the Framework Class
Library (FCL), which includes classes,
interfaces, and protocols for tasks such
as data access, security, networking, and
more. The .NET framework also includes
tools like Visual Studio, which provides
an integrated development environment
(IDE) for building, debugging, and testing
.NET applications. At its core, .NET is a
set of APIs that allow developers to write
code in languages like C#, F#, or VB.NET
and compile it into an intermediate
language called Common Intermediate
Language (CIL), which is then executed by
the .NET runtime environment. This
framework enables developers to create
scalable, efficient, and maintainable
software applications with a wide range of
features and functionalities. Elapsed
time: 13,299 ms
```

OllamaSharp with Microsoft.Extensions.AI

The `Microsoft.Extensions.AI.Abstractions` and `Microsoft.Extensions.AI` packages provide the .NET ecosystem with essential abstractions for integrating AI services into .NET applications and libraries, along with middleware for adding key capabilities.

AI capabilities are rapidly evolving, with common patterns emerging for functionality like “chat,” embeddings, and tool calling. Unified abstractions are crucial for developers to work effectively across different sources. Middleware can add valuable functionality without burdening producers, benefiting consumers immediately.

For example, the Microsoft-defined abstraction `IChatClient` interface allows consumption of language models, whether hosted remotely or running locally. Any .NET package providing an AI client can implement this interface, enabling seamless integration with consuming .NET code.

For example, you might want to use the `OllamaSharp` client when developing locally, and then switch to Azure’s client in production, as shown in the following code:

```
IChatClient client = environment.IsDevelopment
? new OllamaChatClient(...) : new
AzureAIInferenceChatClient(...);
```

Then, in your chat implementation, you can send requests to either client in the same way, as shown in the following code:

```
var response = await chatClient.CompleteAsync(
    "Translate the following text into Pig
    Latin: I love .NET and AI");
WriteLine(response.Message);
```

OllamaSharp is one of the first to implement these abstractions in its `IChatClient`. To do this, simply use the `OllamaApiClient` as `IChatClient`, as shown in the following code:

```
// Reference the  
Microsoft.Extensions.AI.Abstractions package.  
private static IChatClient  
CreateChatClient(Arguments arguments) { if  
(arguments.Provider.Equals("ollama",  
StringComparison.OrdinalIgnoreCase)) return  
new OllamaApiClient(arguments.Uri,  
arguments.Model); else return new  
OpenAIChatClient(new OpenAI.OpenAIClient(  
arguments.ApiKey), arguments.Model); }
```

Now, let's look at an alternative to Ollama for experimenting with local LLMs.

LM Studio

With LM Studio, you can:

- Run LLMs on your laptop, entirely offline.
- Use models through the in-app Chat UI or an OpenAI-compatible local server.
- Download any compatible model files from Hugging Face repositories.
- Discover new and noteworthy LLMs on the app's home page.

To get started with LM Studio:

1. In your preferred browser, navigate to: <https://lmstudio.ai/>.
2. Click the button to download LM Studio for your OS.
3. Once it's downloaded, start the desktop app.
4. Navigate to the model search page and pick a suitable model. Click its

Download button, as shown in *Figure 9.12*:

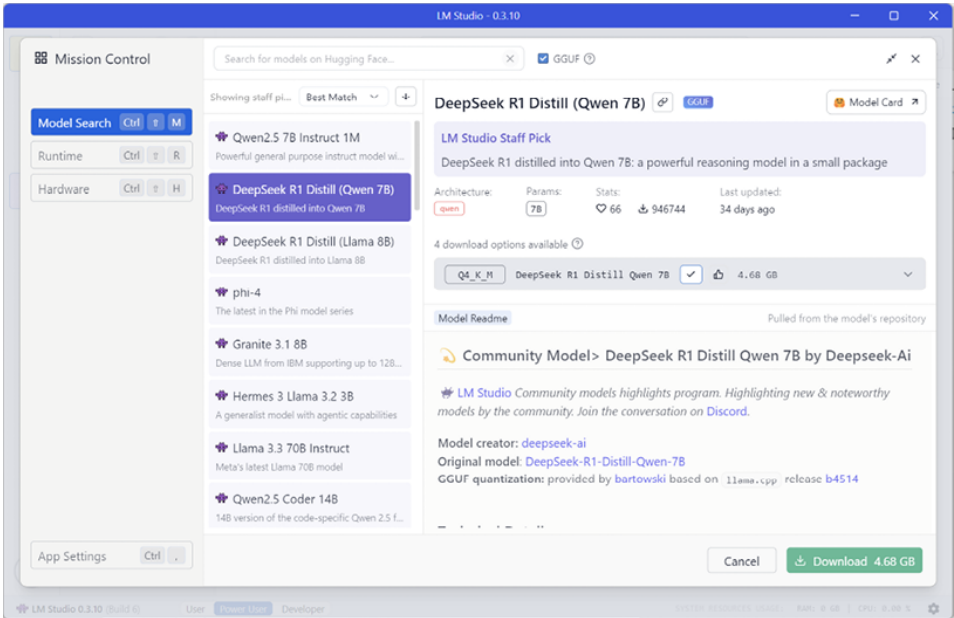


Figure 9.12: The LM Studio model download page for DeepSeek’s R1 7B model

1. Wait for the model to download.
2. Navigate to **Chats**, and at the top of the window, select the model you downloaded.
3. In the **USER** box, enter a prompt, for example: What is .NET in one paragraph?. Note the response, which should be reasonably accurate, as shown in *Figure 9.13*:

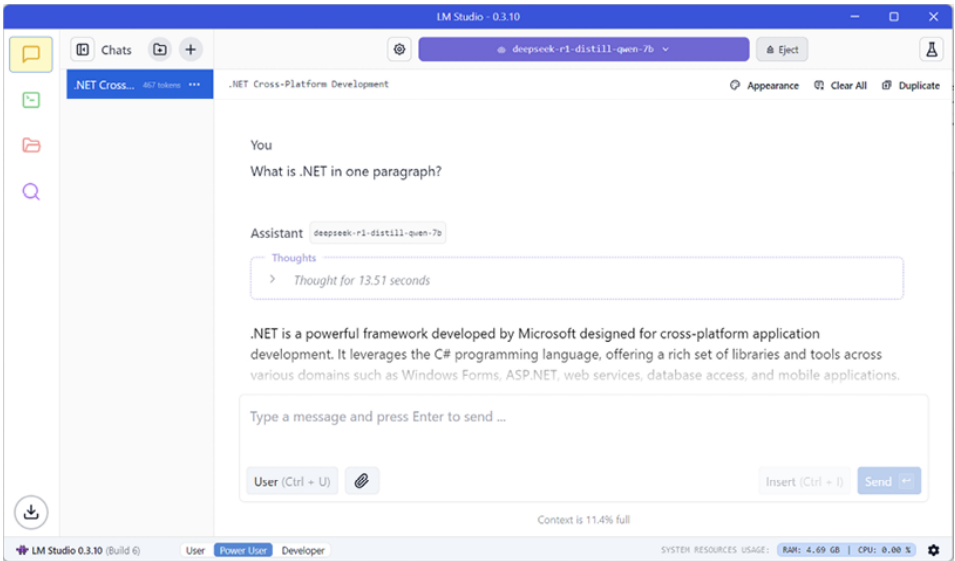


Figure 9.13:LM Studio chat session

1. In the **USER** box, enter a prompt, for example: Who says, "Why is Gamora?", and why?
2. Note that the response may include hallucinations, as shown in the following response:

The phrase "Why is Gamora?" doesn't appear to be widely recognized as an actual quote or line spoken by the character Gamora in any media, including Marvel's Iron Man or other related works. If you encountered this phrasing elsewhere, it might be a misremembered reference to something specific from a particular source or context (like a fan fiction, movie reference, or joke). Could you provide more context? I'd be happy to help further!

3. With advanced models like DeepSeek's R1, it can be useful to expand the **Thinking...** section, as shown in the following output and in *Figure 9.14*:

Thought for 12.84 seconds Alright, so the user asked about who said "Why is Gamora?" and why. Looking back at our conversation history, I had just explained what .NET was in a detailed response. Now, they're asking about a completely different topic—Gamora. Hmm, "Why is Gamora?" That sounds familiar, but I'm not sure where I heard it before. Let me think... It might be from a movie or a show. Maybe Iron Man 3? Gamora is a character from the Avengers, created by Marvel. She's an astrophysicist who has a strong connection with technology. Wait, maybe they're referring to something specific like a line spoken in a particular scene or context. I should check if that phrase exists anywhere. If it's not widely known, perhaps it's from fan fiction or a lesser-known source. Also, considering the user might be confused between different characters or misremembering something. Maybe they saw the character Gamora and associated her with a specific quote without knowing its origin. I should make sure to explain that "Why is Gamora?" isn't a widely recognized line and suggest it's possible they're thinking of a particular source. Offering further help could be useful if they need more information or clarification.

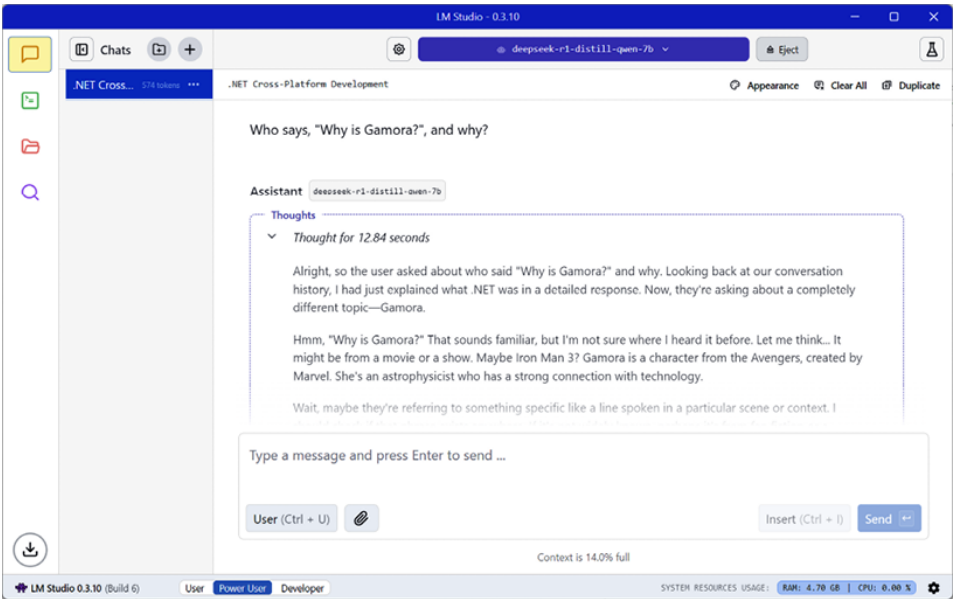


Figure 9.14: What was the model thinking?

1. Now, if I ask the same question to ChatGPT-5.2, I get a much better response:

```
The line "Why is Gamora?" is said by Drax,
a character from the movie Avengers:
Infinity War. This moment occurs during a
scene where the Avengers confront the
Guardians of the Galaxy for the first
time. [cont.]
```

If you test a local model against a cloud model, you will likely see that local models are far behind and much slower, depending on your hardware. However, as we move AI capabilities to the edge, using Copilot+ PCs like Surface Pro 11 and Surface Laptop 7 and their Snapdragon X Elite chips, or if you have a computer with an NVIDIA GPU, this should improve.

You can learn more about Copilot+ PCs at the following link: <https://blogs.microsoft.com/blog/2024/05/20/introducing-copilot-pcs/>.

Practicing and exploring

Test your knowledge and understanding by answering some questions, getting some hands-on practice, and exploring the topics covered in this chapter with deeper research.

Exercise 9.1 – Online-only material

You can read the blog post, *Introducing Microsoft Agent Framework (Preview): Making AI Agents Simple for Every Developer*, at the following link:

<https://devblogs.microsoft.com/dotnet/introducing-microsoft-agent-framework-preview/>

Sample projects that use the MCP C# SDK can be found in its GitHub repository at the following link: <https://github.com/modelcontextprotocol/csharp-sdk/tree/main/samples>

You can read the blog post, *GPT-OSS – A C# Guide with Ollama*, at the following link:

<https://devblogs.microsoft.com/dotnet/gpt-oss-csharp-ollama/>

You can read the blog post, *Build & Leverage MCP Servers in C# for AI-Driven Development*, at the following link:

<https://platform.uno/blog/build-leverage-mcp-servers-in-c-for-ai-driven-development/>:

You can read the blog post, *Building AI-powered Microsoft Copilot with*

SignalR and other open-source tools, at the following link: <https://devblogs.microsoft.com/dotnet/building-ai-powered-bing-chat-with-signalr-and-other-open-source-tools/>

You can read the blog post, *Build Intelligent Applications using ChatGPT & Azure Cosmos DB* at the following link: <https://devblogs.microsoft.com/cosmosdb/chatgpt-azure-cosmos-db/>

Exercise 9.2 – Practice exercises

Here are some ideas for other LLM services that you could build:

1. An AI research assistant trained on academic papers to provide scientific answers with accurate citations.
2. A chatbot to help a .NET developer prepare for an interview.
3. A project chatbot for onboarding new recruits to your team that is trained on the project specification, documentation, and code base so that it can walk the new developer through the project and answer questions about it, instead of the project lead having to answer questions in Slack.
4. A blog article writer who suggests topics for a .NET tip blog article. You choose one, and it helps you write it, then posts it to LinkedIn, X, Threads, and Mastodon for you.

Exercise 9.3 – Test your knowledge

Answer the following questions. If you get stuck, try googling the answers, if necessary, while remembering that if you get totally stuck, the answers are in *Appendix A*:

1. What do you need to call an OpenAI cloud-based LLM?
2. What is Microsoft Agent Framework?
3. What are the benefits of using the AI Chat Web App project template?
4. What does the `AIFunctionFactory.Create` method do?

5. Why is it important to change the default system prompt when using custom functions?
6. What is Hugging Face?

Appendix A, Answers to the Test Your Knowledge Questions, is available to download from the following link: https://github.com/markjprice/apps-services-net10/blob/main/docs/B31467_Appendix%20A.pdf.

Exercise 9.4 – Explore topics

Use the links on the following page to learn more details about the topics covered in this chapter: <https://github.com/markjprice/apps-services-net10/blob/main/docs/book-links.md#chapter-9---building-a-custom-llm-based-chat-service>

Summary

In this chapter, you learned:

- How LLMs work and how to obtain access to one.
- About the concepts around Agent Framework.
- How to extend an LLM with functions.
- How to add session memory and stream results.
- About using local models with tools like Ollama, OllamaSharp, and LM Studio.

In the next chapter, you will learn how to build web services using ASP.NET Core Minimal API, and how to secure and protect them.

Summary

In this chapter, you:

- Were introduced to the app and service technologies that you will learn about in this book.
- Set up your development environment.
- Learned where to look for help.

In the next chapter, you will learn how to build and secure ASP.NET Core Minimal API web services.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow this QR code:

https://packt.link/apps_and_services_dotnet10



Join .NETPro – It's Free

Staying sharp in .NET takes more than reading release notes. It requires real-world tips, proven patterns, and scalable solutions. That's what .NETPro, Packt's new newsletter, is all about.

Scan the QR code or visit the link to subscribe:

<https://landing.packtpub.com/dotnetpronewsletter/>



10

Building and Securing Minimal API Web Services

This chapter is about building and securing web services using ASP.NET Core Minimal API. This includes implementing techniques to protect a web service from attacks as well as authentication and authorization.

This chapter will cover the following topics:

- Understanding web services
- Building web services using ASP.NET Core Minimal API
- Relaxing the same origin security policy using CORS
- Preventing denial of service attacks using rate limiting
- Improving startup time and resources using native AOT
- Understanding identity services

Understanding web services

Before we build a modern web service, we need to cover some background to set the context for this chapter.

Understanding web service acronyms

Although HTTP was originally designed to request and respond with HTML and other resources for humans to look at, it is also good for building services.

Roy Fielding stated in his doctoral dissertation, describing the REST architectural style, that the HTTP standard would be good for building services because it defines the following:

- URIs to uniquely identify resources, such as `https://localhost:5101/api/products/23`.
- Methods for performing common tasks on those resources, such as `GET`, `POST`, `PUT`, and `DELETE`.
- The ability to negotiate the media type of content exchanged in requests and responses, such as XML and JSON. Content negotiation happens when the client specifies a request header such as `Accept: application/xml, */*; q=0.8`. The default response format used by the ASP.NET Core web services is JSON, which means one of the response headers would be `Content-Type: application/json; charset=utf-8`.

Web services use the HTTP communication standard, so they are sometimes called **HTTP services** or **RESTful services**.

Understanding HTTP requests and responses

HTTP defines standard types of requests and standard codes to indicate a type of response. Most of them can be used to implement web services.

Making a GET request and common responses

The most common type of request is `GET`, to retrieve a resource identified by a unique path, with additional options such as what media type is acceptable to set as a request header, like `Accept`, as shown in the following example:

```
GET /path/to/resource Accept: application/json
```

Common responses include success and multiple types of failure, as shown in *Table 10.1*:

Description		
The requester has asked the server to switch protocols, and the server has agreed to do so. For example, it is common to switch from HTTP to WebSockets (WS) for more efficient communication.		
This is used to convey hints that help a client make preparations to process the final response. For example, the server might send the following response before then sending a normal 200 OK response for a web page that uses a stylesheet and JavaScript file: HTTP/1.1 103 Early Hints Link: </style.css>; rel=preload; as=style Link: </script.js>; rel=preload; as=script		
The OK was correctly formed, and the resource was successfully found, serialized into an acceptable media type, and then returned in the response body. The response headers specify Content-Type, Content-Length, and Content-Encoding, for example, GZIP.		
Over time, a web service may change its resource model, including the path used to identify an existing resource. The web service can indicate the new path by returning this status code and a response header named Location that contains the new path.		
This is the same as 301.		
If the request includes the If-Modified-Since header, then the web service can respond with this status code. The response body is empty because the client should use its cached copy of the resource.		
The requested resource has been temporarily moved to the URL in the Location header. The browser should make a new request using that URL. For example, this is what happens if you enable		

	Use <code>HttpResponseRedirect</code> and a client makes an HTTP request	
	The request was invalid, for example, it used a path for a product using an integer ID where the ID value is missing	
	The request was valid and the resource was found, but the client did not supply credentials or is not authorized to access that resource. Re-authenticating may enable access, for example, by adding or changing the <code>Authorization</code> request header	
	The request was valid and the resource was found, but the client is not authorized to access that resource. Re-authenticating will not fix the issue	
	The request was valid, but the resource was not found. The resource may be found if the request is repeated later. To indicate that a resource will never be found, return <code>410 Gone</code>	
	The request has an <code>Accept</code> header that only lists media types that the web service does not support. For example, the client requests JSON, but the web service can only return XML	
	The request couldn't be completed because it conflicts with the current state of the resource. For example, a canceled payment cannot be canceled again, or when trying to upload or update a resource that already exists in a way that violates uniqueness, such as creating a user with a username that's already taken	
	The client has sent too many requests in a given amount of time. The client should reduce the rate due to this rate limiting. A <code>Retry-After</code> header may be included in this response to indicate how long a client should wait before making another request	
	A website hosted in the USA might return a 515 for requests coming from Europe to avoid having to comply with the General Data Protection Regulation (GDPR) . The number was chosen as a reference to the novel <i>Fahrenheit 451</i> , in which books are banned and burned	
	The request was valid, but something went wrong on the server side while processing the request. Trying again later might work	
	The web service is busy and cannot handle the request. Trying again later might work	

408 Request Timeout	A server acting as a gateway or proxy didn't get a timely response from the upstream server it needed to access to complete the request.
---------------------	--

Table 10.1: Common HTTP status code responses to the GET method

Making other requests, such as POST, PUT, and DELETE, and common responses

Other common types of HTTP requests include POST, PUT, PATCH, and DELETE, which create, modify, or delete resources.

To create a new resource, you might make a POST request with a body that contains the new resource, as shown in the following code:

```
POST /path/to/resource Content-Length: 123
Content-Type: application/json { ... }
```

To create a new resource or update an existing resource, you might make a PUT request with a body that contains a whole new version of the existing resource. If the resource does not exist, it is created, or if it does exist, it is replaced (sometimes called an **upsert** operation), as shown in the following code:

```
PUT /path/to/resource Content-Length: 123
Content-Type: application/json { ... }
```

To update an existing resource more efficiently, you might make a PATCH request with a body that contains an object with only the properties that need changing, as shown in the following code:

```
PATCH /path/to/resource Content-Length: 123
Content-Type: application/json-patch+json {
  ... }
```

You can learn more about JSON Patch at the following link: https://en.wikipedia.org/wiki/JSON_Patch.

```
DELETE /path/to/resource
```

```
DELETE /path/to/resource
```

As well as the responses shown in the preceding table for a GET request, all the types of requests that create, modify, or delete a resource have additional possible common responses, as shown in *Table 10.2*:

HTTP Status Code	Meaning	Response Body
201	Created	The new resource was created successfully, the response header named Location contains its path, and the response body contains the newly created resource. Immediately GET-ing the resource should return 200.
202	Accepted	The new resource cannot be created immediately, so the request is queued for later processing, and immediately GET-ing the resource might return 404. The body can contain a resource that points to some form of status checker or an estimate of when the resource will become available.
204	No Content	This is commonly used in response to a DELETE request, since returning the resource in the body after deleting it does not usually make sense! It is also sometimes used in response to POST, PUT, or PATCH requests if the client does not need to confirm that the request was processed correctly.
405	Method Not Allowed	This is returned when the request used a method that is not supported. For example, a web service designed to be read-only may explicitly disallow PUT, DELETE, and so on.
415	Unsupported Media Type	This is returned when the request body uses a media type that the web service cannot handle. For example, if the body contains a resource in XML format, but the web service can only process JSON.

Documenting web services with Swagger, OpenAPI, and Swashbuckle

Let's start by explaining some terminology related to documenting web services:

- **Swagger:** Originally, Swagger was a framework for describing, documenting, and trying out REST APIs. It included tools such as the Swagger UI and Swagger Editor. However, Swagger evolved into the **OpenAPI Specification (OAS)**, which is now the industry standard for defining RESTful APIs in a machine-readable format (YAML or JSON).
- **OpenAPI:** This is the formalized specification that defines how to describe and document REST APIs. The OAS provides a standardized way to describe API endpoints, request/response models, authentication, and more. OpenAPI is maintained by the **OpenAPI Initiative (OAI)** under the Linux Foundation.
- **Swashbuckle:** This is one of many .NET libraries that automatically generate OpenAPI documentation for ASP.NET Core Web API web services. It implements a Swagger UI, allowing you to visualize and test API endpoints directly in the browser. It became particularly popular after Microsoft included it by default in the ASP.NET Core Web API project template.

It is easy to confuse Swashbuckle and Swagger because they start with similar letters: *Swa*. Try to remember that you should avoid using both terms because they have better alternatives:

- When talking about the specification that defines how to describe and document REST APIs, replace Swagger with OpenAPI
- When referencing a package to describe and document a web service project, replace Swashbuckle with a more modern package such as Scalar or NSwag

Recently, the ASP.NET Core team has changed how these technologies are used in ASP.NET Core projects:

- In ASP.NET Core 8 and earlier, the third-party `Swashbuckle.AspNetCore` package was used to generate an OpenAPI JSON document to formally describe the web service. Swashbuckle includes a Swagger UI that provides an interactive web page to explore and try out API endpoints. But the Swashbuckle package is third-party, so it is out of the control of Microsoft, and it has not been maintained well enough by its owner for Microsoft to want to use it.
- In ASP.NET Core 9 and later, the first-party `Microsoft.AspNetCore.OpenApi` package is used to generate an OpenAPI JSON document to formally describe the web service. But this package does not provide an interactive UI for trying out the web service.

You can learn more about how to use OpenAPI documentation at the following link: <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/openapi/using-openapi-documents>.

Building web services using ASP.NET Core Minimal API

Now let's look at how to build modern web services in practice.

ASP.NET Core Minimal API web service projects

Traditionally, you use the **ASP.NET Core Web API** / `dotnet new webapi` project template. This allows the creation of a web service implemented using either controllers or the newer Minimal API.

Warning! With .NET 6 and .NET 7, the `dotnet new webapi` command creates a service implemented using controllers. With .NET 6 and .NET 7, to implement the service using the Minimal API, you need to add the `--use-minimal-apis` switch to the command. Using .NET 8 or later, the `dotnet new webapi` command creates a service implemented using the Minimal API. To implement the service using controllers, you need to add the `--use-controllers` switch.

In older versions of ASP.NET Core, you would build a web service using controllers with an action method for each endpoint, a bit like building a website with ASP.NET Core MVC using controllers and models but without the views. Since .NET 6, you have another, often better, choice: **ASP.NET Core Minimal API**.

Benefits of Minimal API-based web services

In earlier versions of ASP.NET Core, implementing even a simple web service required a lot of boilerplate code compared to alternative web development platforms. For example, a minimal `Hello World` web service implementation that has a single endpoint that returns plain text could be implemented using **Express.js** in just nine lines of code, as shown in the following code:

```
const express = require('express') const app =
express() const port = 3000 app.get('/', (req,
res) => { res.send('Hello World!') })
app.listen(port, () => { console.log(`Example
```



```
app listening on port ${port}`) })
```

With ASP.NET Core 5 or earlier, that would require more than fifty lines of code!

The equivalent of using ASP.NET Core 6 or later with ASP.NET Core Minimal API is now only five lines of code and six lines of configuration, as shown in the following two code blocks:

```
int port = 3000; var app =  
WebApplication.Create(); app.MapGet("/", () =>  
"Hello World!"); Console.WriteLine($"Example  
app listening on port {port}"); await  
app.RunAsync($"https://localhost:{port}/");
```

The platform is specified in the project file, and the implicit `using` statements SDK feature does some heavy lifting. It is enabled by default, as shown highlighted in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk.Web">  
<PropertyGroup> <TargetFramework>net10.0</  
TargetFramework> <ImplicitUsings>enable</  
ImplicitUsings> </PropertyGroup> </Project>
```

Good practice: Another benefit of Minimal API web services is that they do not use dynamically generated code, unlike controller-based Web APIs. This allows them to use native AOT to produce smaller, faster services that are better for implementing and hosting microservices in containers. We will cover native AOT with Minimal API later in this chapter. Whenever

possible, implement your web services using Minimal API endpoints instead of controllers.

So those are the benefits of using Minimal API compared to controller-based Web API web services. Now let's see how Minimal API defines route mappings that associate the path of an incoming HTTP request to the appropriate endpoint to process that request.

Understanding Minimal API route mappings

The `WebApplication` instance has methods that you can call to map a route to a lambda expression or statement:

- `MapGet`: Map a route to a `GET` request to retrieve an entity.
- `MapPost`: Map a route to a `POST` request to insert an entity.
- `MapPut`: Map a route to a `PUT` request to update an entity.
- `MapPatch`: Map a route to a `PATCH` request to update an entity.
- `MapDelete`: Map a route to a `DELETE` request to delete an entity.
- `MapMethods`: Map a route to any other HTTP method or methods, for example, `CONNECT` or `HEAD`.

For example, you might want to map an HTTP `GET` request for the relative path `api/customers` to a delegate defined by a lambda expression or a function that returns a JSON document containing a list of customers, and equivalent mappings to insert and delete, as shown in the following code:

```
app.MapGet("api/customers", GetCustomers);  
app.MapPost("api/customers", InsertCustomer);  
app.MapDelete("api/customers/{id}",  
DeleteCustomer);
```

You might want to map an HTTP `CONNECT` request for the relative path

api/customers to a lambda statement block, as shown in the following code:

```
app.MapMethods("api/customers", new[] {  
    "CONNECT" }, () => { // Do something. });
```

If you have multiple endpoints that share a common relative path, then you can define a **route group**. The `MapGroup` method was introduced in .NET 7:

```
RouteGroupBuilder group = app.MapGroup("api/  
customers"); group.MapGet("/", GetCustomers)  
.MapGet("/{id}", GetCustomerById)  
.MapPost("/", InsertCustomer) .MapDelete("/  
{id}", DeleteCustomer);
```

You can learn more about mapping routes at the following link: <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/minimal-apis/route-handlers>.

Understanding MapWhen and MapGroup

Let's briefly review some related methods for mapping.

MapWhen

`MapWhen` conditionally branches the entire request pipeline based on a predicate. If the predicate returns `true`, the request is handled by a separate middleware pipeline. If not, it falls through to the normal pipeline.

Think of it as “if this request looks like X, send it down this completely different path”.

This runs before routing, so it is not about endpoints. It is about middleware flow.

Use it to:

- Split traffic by hostname, headers, or path prefixes
- Route legacy behavior without polluting your main pipeline
- Handle special cases like health probes, admin tunnels, or internal APIs

Here's an example that branches based on a request header:

```
app.MapWhen( context =>
context.Request.Headers.ContainsKey("X-
Internal-Request"), internalApp => {
internalApp.Run(async context => { await
context.Response.WriteAsync("Internal
pipeline"); }); });
```

If the header `X-Internal-Request` is present, the request never touches the normal public pipeline that processes all other requests.

Here's an example that branches by path prefix:

```
app.MapWhen( ctx =>
ctx.Request.Path.StartsWithSegments("/
legacy"), legacyApp => { legacyApp.Run(async
ctx => { await ctx.Response.WriteAsync("Legacy
system"); }); });
```

This is cleaner than using `if` statements inside middleware and avoids unnecessary routing work.

MapGroup

`MapGroup` is for organizing endpoints. It groups routes under a shared prefix

and lets you apply shared metadata, filters, authorization, and conventions.

It works with endpoint routing, not middleware branching.

Think “folder for routes with shared rules.”

Use it to:

- Group APIs by feature or version
- Apply authorization once instead of per endpoint
- Keep Minimal APIs readable as they grow

Here’s an example that groups with a route prefix:

```
var products = app.MapGroup("/products");
products.MapGet("/", () => "All products");
products.MapGet("/{id:int}", (int id) =>
    $"Product {id}");
```

The following requests would be processed by this grouping:

- /products → All products
- /products/42 → Product 42

Here’s an example that groups with shared authorization and filters:

```
var admin = app.MapGroup("/admin")
    .RequireAuthorization(); admin.MapGet("/
users", () => "Admin users"); admin.MapPost("/
users", () => "Create user");
```

A benefit is that you avoid repeating `.RequireAuthorization()` everywhere, which makes the intent obvious and the code cleaner.

If you ever find yourself trying to use `MapWhen` just to organize routes, you are doing it wrong. If you are using `MapGroup` to separate radically different request behavior, that is also wrong. They solve different problems, and they do

not overlap.

Now, let's look at how to control how parameters are set for an endpoint.

Understanding parameter mapping

The delegate can have parameters defined that can be set automatically.

Although most mappings can be configured without being explicitly specified, you can optionally use attributes to define where ASP.NET Core Minimal API web services should set the parameter values from:

- `[FromServices]`: The parameter will be set from the registered dependency services.
- `[FromRoute]`: The parameter will be set from a matching named route segment.
- `[FromQuery]`: The parameter will be set from a matching named query string parameter.
- `[FromBody]`: The parameter will be set from the body of the HTTP request.

For example, to update an entity in a database, you would need a database context to be retrieved from the registered dependency services, an identifier passed as a query string or route segment, and the new entity in the body of the request, as shown in the following code:

```
app.MapPut("api/customers/{id}", async (
    [FromServices] NorthwindContext db,
    [FromRoute] string id, // or [FromQuery]
    string id, [FromBody] Customer customer) => {
    Customer? existingCustomer = await
    db.Customers.FindAsync(id); ... });
```

Understanding return values

A minimal API service can return data in some common formats, as shown in *Table 10.3*:

Text		
Plain text	<code>"Hello world!"</code>	
	<code>() => Results.Text("Hello World!")</code>	
	<code>() => TypedResults.Text("Hello World!")</code>	
JSON document	<code>new { FirstName = "Bob", LastName = "Jones" }</code>	
	<code>() => Results.Json(new { FirstName = "Bob", LastName = "Jones" })</code>	
	<code>() => TypedResults.Json(new { FirstName = "Bob", LastName = "Jones" })</code>	
Result with status code	<code>Ok(new { FirstName = "Bob", LastName = "Jones" })</code>	
	<code>() => TypedResults.NoContent()</code>	
	<code>() => TypedResults.Redirect("new/path")</code>	
	<code>() => TypedResults.NotFound()</code>	
	<code>() => TypedResults.BadRequest()</code>	
	<code>() => TypedResults.Problem()</code>	
	<code>() => TypedResults.StatusCode(405)</code>	
File	<code>File("/path/filename.ext")</code>	

Table 10.3: Examples of minimal API return values

Good practice: `TypedResults` improves type safety and helps with testing. Since its methods return concrete `IResult` implementations rather than a generic `IResult`, it's easier to assert on actual result types in unit tests. If you care about generating good metadata for documentation, using `TypedResults` helps because the return types embed metadata about possible responses.

Returning multiple typed outcomes

With `TypedResults`, you can declare the possible return types up front, and the compiler enforces them, as shown in the following code:

```
app.MapGet("/api/orders/{id}",
Results<Ok<Order>, NotFound> (int id) => { if
(!Database.TryGetOrder(id, out var order)) {
return TypedResults.NotFound(); } return
TypedResults.Ok(order); });
```

Without `TypedResults`, developers often return raw `IResult`, which makes testing more awkward and provides weaker metadata for OpenAPI.

Improving the OpenAPI document for a Minimal API web service

You can call additional methods as many times as you need to specify what return types and status codes can be expected from an endpoint, for example:

- `Produces<T>(StatusCodes.Status200OK):` When successful, this route returns a response containing a type `T` and status code `200`.
- `Produces(StatusCodes.Status404NotFound):` When no match for the route is found, this route returns an empty response and status code `404`.

Designing an ASP.NET Core Minimal API project

The API for this web service is defined as shown in *Table 10.4*:

Endpoint	Request	Response	Media type	HTTP status code	Authentication	Authorization	Cache	Comments
GET /api/orders/{id}		Ok<Order>	application/json	200				
GET /api/orders/{id}		NotFound		404				
POST /api/orders		Ok<Order>	application/json	201				
DELETE /api/orders/{id}		Ok<Order>		200				

None	/in-stock	Product objects								
None	/on-manifest	Product objects								
None	/discontinued	Product objects								
None	/products/{id}									
None	/products/objects that contain the name									
None	/product (no id value)									
None	/products/{id}									
None	/products/{id}									

Table 10.4: API methods implemented by the example project

Setting up an ASP.NET Core Minimal API project

Now that we have designed the endpoints, we can create the ASP.NET Core Minimal API web service project that we will later protect using various techniques like rate limiting, CORS, and authentication and authorization.

Let’s go:

1. Use your preferred code editor to add a Web API project to a new ModernServices solution, as defined in the following list (make sure to scroll down to see all options):
- Project template: **ASP.NET Core Web API / webapi**
 - Location: `c:\apps-services-net10\`
 - Solution file and folder: `ModernServices`
 - Project file and folder:
`Northwind.WebApi.Service`
 - **Authentication type:** **None**
 - **Configure for HTTPS:** **Selected**
 - **Enable container support:** **Cleared**
 - **Enable OpenAPI support:** **Selected**
 - **Do not use top-level statements:** **Cleared**

- **Use controllers:** Cleared
- **Enlist in Aspire orchestration:** Cleared

To create a Web API project using Minimal API with `dotnet new` for pre-.NET 8 SDKs, you must use either the `-minimal` switch or the `--use-minimal-apis` switch. For .NET 8 and later SDKs, Minimal API is the default, and to use controllers, you must specify the `--use-controllers` or `-controllers` switch.

Warning! If you are using Rider, its user interface might not yet have an option to create a web API project using Minimal API. I recommend creating the project using `dotnet new` and then adding the project to your solution.

1. In the `ModernServices` solution, add the two existing projects for the Northwind entity models and database context, as shown highlighted in the following markup:

```
<Solution> <Project Path="..\ModernApps/  
Northwind.DataContext/  
Northwind.DataContext.csproj" /> <Project  
Path="..\ModernApps/  
Northwind.EntityModels/  
Northwind.EntityModels.csproj" /> <Project  
Path="Northwind.WebApi.Service/  
Northwind.WebApi.Service.csproj" /> </  
Solution>
```

2. In the `Northwind.WebApi.Service.csproj` project file, add a project reference to the Northwind database context project, delete the version attribute from the package reference, and treat warnings as errors, as shown in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup> <TargetFramework>net10.0</
  TargetFramework> <Nullable>enable</
  Nullable> <ImplicitUsings>enable</
  ImplicitUsings>
  <TreatWarningsAsErrors>true</
  TreatWarningsAsErrors> </PropertyGroup>
  <ItemGroup> <PackageReference
  Include="Microsoft.AspNetCore.OpenApi" />
  </ItemGroup> <ItemGroup> <ProjectReference
  Include= "..\..\ModernApps
  \Northwind.DataContext
  \Northwind.DataContext.csproj" /> </
  ItemGroup> </Project>
```

3. Build the `Northwind.WebApi.Service` project.
4. In the `Properties` folder, in `launchSettings.json`, modify the `applicationUrl` of the profile named `https` to use port 5101, as shown highlighted in the following configuration:

```
"profiles": { ... "https" :{
  "commandName": "Project",
  "dotnetRunMessages": true,
  "launchBrowser": false, "applicationUrl"
  "https://localhost:5101;http://
  localhost:5100":, "environmentVariables":
  { "ASPNETCORE_ENVIRONMENT": "Development"
  }
}
```

5. In `Program.cs`, delete the statements about the weather service and replace them with statements to import the namespace to use the `AddNorthwindContext` extension method, and then call it to add the `NorthwindContext` to configured services, as shown highlighted in the following code:

```
using Northwind.EntityModels; // To use  
the AddNorthwindContext method. var  
builder =  
WebApplication.CreateBuilder(args); // Add  
services to the container. // Learn more  
about configuring OpenAPI at https://  
aka.ms/aspnet/openapi  
builder.Services.AddOpenApi();  
builder.Services.AddNorthwindContext();  
var app = builder.Build(); // Configure  
the HTTP request pipeline. if  
(app.Environment.IsDevelopment()) {  
    app.MapOpenApi(); }  
app.UseHttpsRedirection(); app.Run();
```

6. In the `Northwind.WebApi.Service` project, add a new folder named `Extensions`.
7. In the `Extensions` folder, add a new class file named `WebApplication.Extensions.cs`.

Good practice: Instead of cluttering your `Program.cs` file with hundreds of lines of code, define extension methods for the common types that are configured in a Minimal API web service, like `WebApplication` and `IServiceCollection`.

1. In `WebApplication.Extensions.cs`, import namespaces for controlling HTTP results, binding a parameter to a dependency service, and working with Northwind entity models, and then define an extension method for the `WebApplication` class to configure responses to all the HTTP GET requests documented in our API table, as shown in the following code:

To save you time and since this is a long file, you might want to copy the complete file from the following link: <https://github.com/markjprice/apps-services-net10/blob/main/code/ModernServices/Northwind.WebApi.Service/WebApplication.Extensions.cs>

```
using Microsoft.AspNetCore.Http.HttpResults;
// To use Results<T, T>. using
Microsoft.AspNetCore.Mvc; // To use
[FromServices] and so on. using
Northwind.EntityModels; // To use
NorthwindContext, Product. namespace
Northwind.WebApi.Service.Extensions; public
static class WebApplicationExtensions { public
static WebApplication MapGets( this
WebApplication app, int pageSize = 10) {
app.MapGet("/", () => "Hello World!");
app.MapGet("api/products", ( [FromServices]
NorthwindContext db, [FromQuery] int? page) =>
db.Products .Where(p => p.UnitsInStock > 0 &&
!p.Discontinued) .OrderBy(product =>
product.ProductId) .Skip(((page ?? 1) - 1) *
pageSize) .Take(pageSize) )
```

```

.WithName("GetProducts")
.WithDescription("Gets products with
UnitsInStock > 0 and Discontinued = false.")
.WithSummary("Gets in-stock products that are
not discontinued.")
.Produces<Product[]>(StatusCodes.Status200OK);
app.MapGet("api/products/outofstock",
([FromServices] NorthwindContext db) =>
db.Products .Where(p => p.UnitsInStock == 0 &&
!p.Discontinued) )
.WithName("GetProductsOutOfStock")
.WithDescription("Gets products with
UnitsInStock = 0 and Discontinued = false.")
.WithSummary("Gets out-of-stock products that
are not discontinued.")
.Produces<Product[]>(StatusCodes.Status200OK);
app.MapGet("api/products/discontinued",
([FromServices] NorthwindContext db) =>
db.Products.Where(product =>
product.Discontinued) )
.WithName("GetProductsDiscontinued")
.WithDescription("Gets products that are
discontinued.") .WithSummary("Gets all
discontinued products.")
.Produces<Product[]>(StatusCodes.Status200OK);
app.MapGet("api/products/{id:int}", async
Task<Results<Ok<Product>, NotFound>> (
[FromServices] NorthwindContext db,
[FromRoute] int id) => await
db.Products.FindAsync(id) is Product product ?
TypedResults.Ok(product) :
TypedResults.NotFound())
.WithName("GetProductById")
.WithDescription("Gets a product by its ID.")
.WithSummary("Gets a product by its ID.")

```

```

.Produces<Product>(StatusCodes.Status200OK)
.Produces(StatusCodes.Status404NotFound);
app.MapGet("api/products/{name}", (
[FromServices] NorthwindContext db,
[FromRoute] string name) =>
db.Products.Where(p =>
p.ProductName.Contains(name)))
.WithName("GetProductsByName")
.WithDescription("Gets products by their
name.") .WithSummary("Gets products by their
name.")
.Produces<Product[]>(StatusCodes.Status200OK);
return app; } }

```

1. In `WebApplication.Extensions.cs`, define an extension method for the `WebApplication` class to configure a response to the HTTP POST request documented in the API table, as shown in the following code:

```

public static WebApplication MapPosts(this
WebApplication app) { app.MapPost("api/
products", async ([FromBody] Product
product, [FromServices] NorthwindContext
db) => { db.Products.Add(product); await
db.SaveChangesAsync(); return
Results.Created($"api/products/
{product.ProductId}", product);
}).Produces<Product>(StatusCodes.Status201Created);
return app; }

```

2. In `WebApplication.Extensions.cs`, define an extension method for the `WebApplication` class to configure a response to the HTTP PUT request documented in the API table, as shown in the

following code:

```
public static WebApplication MapPuts(this
WebApplication app) { app.MapPut("api/
products/{id:int}", async ( [FromRoute]
int id, [FromBody] Product product,
[FromServices] NorthwindContext db) => {
Product? foundProduct = await
db.Products.FindAsync(id); if
(foundProduct is null) return
Results.NotFound();
foundProduct.ProductName =
product.ProductName;
foundProduct.CategoryId =
product.CategoryId;
foundProduct.SupplierId =
product.SupplierId;
foundProduct.QuantityPerUnit =
product.QuantityPerUnit;
foundProduct.UnitsInStock =
product.UnitsInStock;
foundProduct.UnitsOnOrder =
product.UnitsOnOrder;
foundProduct.ReorderLevel =
product.ReorderLevel;
foundProduct.UnitPrice =
product.UnitPrice;
foundProduct.Discontinued =
product.Discontinued; await
db.SaveChangesAsync(); return
Results.NoContent();
}).Produces(StatusCodes.Status404NotFound)
.Produces(StatusCodes.Status204NoContent);
return app; }
```


3. In `WebApplication.Extensions.cs`, define an extension method for the `WebApplication` class to configure a response to the HTTP DELETE request documented in the API table, as shown in the following code:

```
public static WebApplication
MapDeletes(this WebApplication app) {
    app.MapDelete("api/products/{id:int}",
        async ( [FromRoute] int id, [FromServices]
        NorthwindContext db) => { if (await
        db.Products.FindAsync(id) is Product
        product) { db.Products.Remove(product);
        await db.SaveChangesAsync(); return
        Results.NoContent(); } return
        Results.NotFound();
    }).Produces(StatusCodes.Status404NotFound)
    .Produces(StatusCodes.Status204NoContent);
    return app; }
```

4. In `Program.cs`, import the namespace to use the extension methods you just defined, as shown in the following code:

```
using Northwind.WebApi.Service.Extensions;
// To use MapGets and so on.
```

5. In `Program.cs`, before the call to `app.Run()`, call your custom extension methods to map GET, POST, PUT, and DELETE requests, noting that you can override the default page size of 10 entities when requesting all products, as shown in the following code:

```
app.MapGets() // Default pageSize: 10.
.MapPosts() .MapPuts() .MapDeletes();
```

6. In `Program.cs`, make sure the last statement in the file runs the web app, as shown in the following code:

```
app.Run();
```

Testing web services using OpenAPI

Now we can start the web service, see its documentation using OpenAPI, and perform basic manual testing:

1. If your database server is not running (for example, because you are hosting it in Docker, a virtual machine, or in the cloud), then make sure to start it.
2. Start the web service project using the `https` profile:
 - If you are using Visual Studio, select the **https** profile in the drop-down list and then navigate to **Debug | Start Without Debugging** or press `Ctrl + F5`.
 - If you are using VS Code, enter the command `dotnet run --launch-profile https`.

On Windows, if prompted to do so, you will have to set Windows Defender Firewall to allow access to your local web service.

1. Manually start a web browser and navigate to the OpenAPI document: <https://localhost:5101/openapi/v1.json>.
2. In your web browser, note the OpenAPI JSON document, as shown in *Figure10.1*:

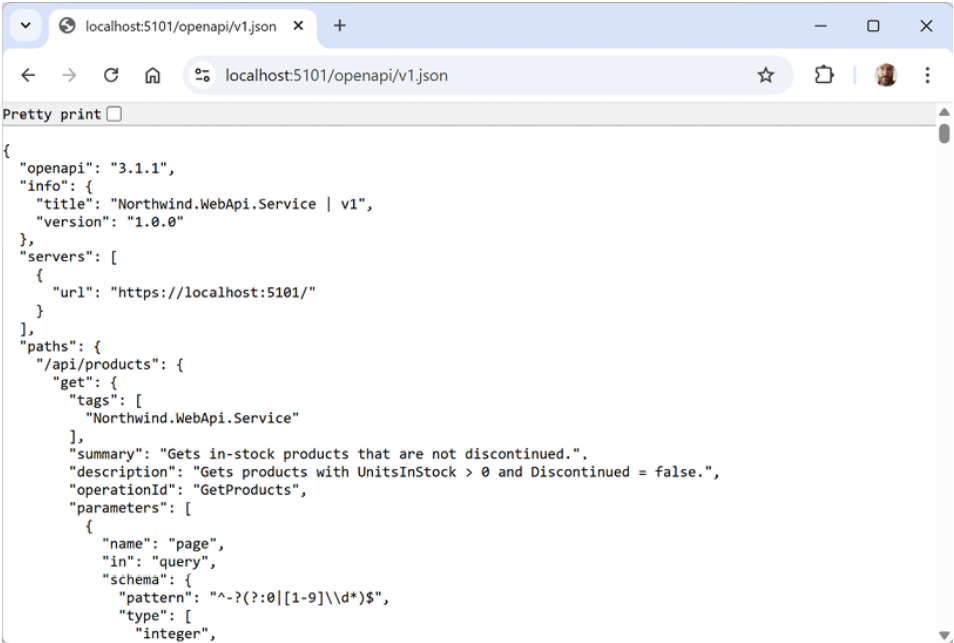


Figure 10.1: OpenAPI JSON document for the Northwind web service

1. Navigate to `/api/products`, and note the response includes the first ten products that are in stock and not discontinued: 1, 2, 3, 4, 6, 7, 8, 10, 11, and 12, as shown in Figure 10.2:

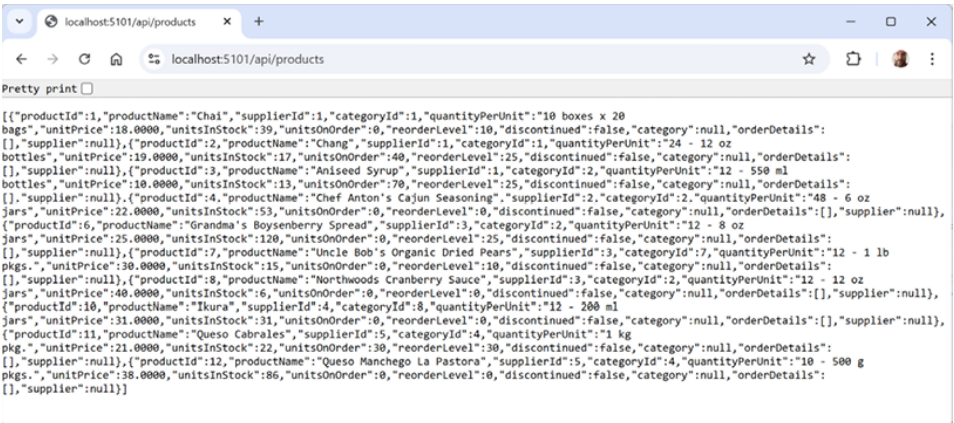


Figure 10.2: First page of ten products from the Northwind web service

1. Navigate to `/api/products?page=3`, and note that the response includes the third page of ten products that are in stock and are not discontinued: 25, 26, 27, 30, 32, 33, 34, 35, 36, and 37.
2. Navigate to `/api/products/outofstock` and note it returns one product, **31 Gorgonzola Telino**, which has zero units in stock and is not discontinued.
3. Navigate to `/api/products/discontinued` and note it returns eight products, 5, 9, 17, 24, 28, 29, 42, and 53, which all have their `Discontinued` properties set to `true`.
4. Navigate to `/api/products/77` and note the response contains the product named **Original Frankfurter grüne Soße**, as shown in the following JSON document:

```
{ "productId": 77, "productName":  
  "Original Frankfurter grüne Soße",  
  "supplierId": 12, "categoryId": 2,  
  "quantityPerUnit": "12 boxes",  
  "unitPrice": 13, "unitsInStock": 32,  
  "unitsOnOrder": 0, "reorderLevel": 15,  
  "discontinued": false, "category": null,  
  "supplier": null, "orderDetails": [] }
```

5. Navigate to `/api/products/man` and note the response contains the products named **Queso Manchego La Pastora** and **Manjimup Dried Apples**.
6. Close the browser and shut down the web server.

Implementing Scalar for documentation

Scalar is gaining popularity with ASP.NET Core developers.

Let's add a modern third-party package to provide a user interface to try out our web service:

1. In the `Northwind.WebApi.Service.csproj` project file, add a package reference for Scalar's integration with ASP.NET Core, as shown in the following markup:

```
<PackageReference  
  Include="Scalar.AspNetCore" />
```

2. Build the `Northwind.WebApi.Service` project to restore packages.
3. In `Program.cs`, import the Scalar namespace, as shown in the following code:

```
using Scalar.AspNetCore; // To use  
  MapScalarApiReference method.
```

4. In `Program.cs`, add a statement so that when in the development environment, you enable Scalar's API reference capability, as shown highlighted in the following code:

```
if (app.Environment.IsDevelopment()) {  
  app.MapOpenApi();  
  app.MapScalarApiReference(); }
```

You could integrate Scalar with Swashbuckle and NSwag, as well as Microsoft's OpenAPI implementation.

1. Start the `Northwind.WebApi` web service project using the `https` launch profile.
2. Navigate to <https://localhost:5101/scalar/v1>, and note

the Scalar UI shows a list of endpoints such as `/customers/{id}` GET and `/customers/{id}` DEL in the left navigation, as shown in *Figure 10.3*:

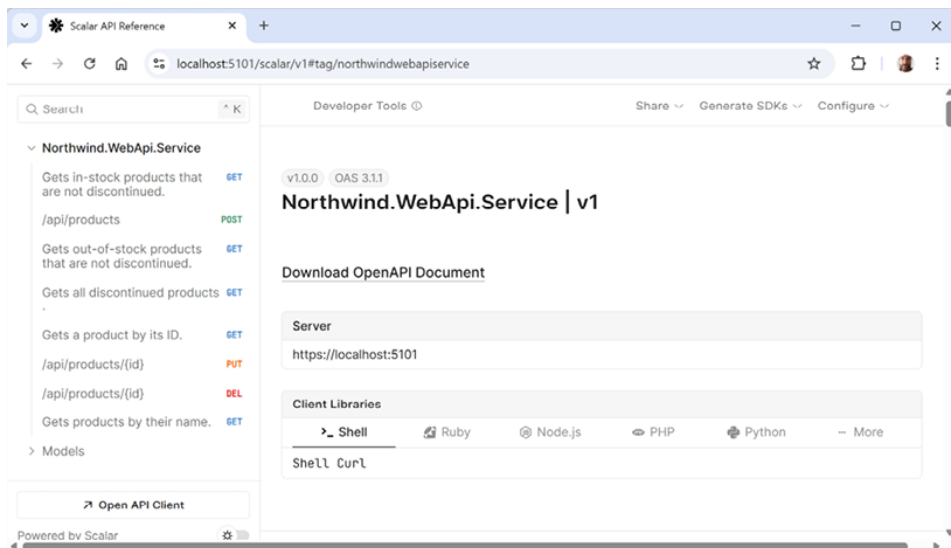


Figure 10.3: Scalar listing all endpoints in a web service

1. Click one of the endpoints and try calling it with appropriate parameter values.
2. Close the browser and shut down the web server.

You can learn more about Scalar at the following link: <https://scalar.com/>.

Testing web services with code editor tools

Using a browser to test web services can quickly get clumsy. A better tool is either the VS Code extension named **REST Client** or the **Endpoints Explorer** and `.http` file support available with Visual Studio version 17.6 or later.

You can learn about Visual Studio and its HTTP editor at the following link: <https://learn.microsoft.com/en-us/aspnet/core/test/http-files>.

Rider has a similar tool window named **Endpoints**.

If you are using Rider, you can read about its tools for HTTP files at the following link: https://www.jetbrains.com/help/rider/Http_client_in_product_code_editor.html. It is slightly different from the other two code editors. In particular, how Rider handles setting variables is more awkward, as shown at the following link: https://www.jetbrains.com/help/rider/Exploring_HTTP_Syntax.html#example-working-with-environment-files. You might prefer to use VS Code with the REST Client extension for this section.

Let's see how these tools enable us to test a web service:

1. Make sure you have the web service testing tools installed:

- If you are using Visual Studio, make sure you have version 17.6 or later.
- If you are using VS Code, make sure you have installed the REST Client extension by Huachao Mao (`humao.rest-client`).
- If you are using Visual Studio, navigate to **View | Other Windows | Endpoints Explorer**, and note the current project is scanned for potential Web API endpoints, as shown in *Figure 10.4*.

5. In your preferred code editor, start the `Northwind.WebApi.Service` project using the `https` profile (if it is not already running) and leave it running.
6. In the `apps-services-net10` folder, if it does not already exist, create an `HttpRequests` folder.
7. In the `HttpRequests` folder, create a file named `webapi-get-products.http` and modify its contents to declare a variable to hold the base address of the Web API service products endpoint and a request to get the first page of ten products, as shown in the following code:

```
### Configure a variable for the web  
service base address. @base_address =  
https://localhost:5101/api/products/ ###  
Get first page of 10 products that are in  
stock and not discontinued. GET  
{{base_address}}
```

Good practice: REST Client does not require `GET` at the beginning of a request because it will assume `GET` as the default. But at the time of writing, Visual Studio's HTTP editor requires `GET` to be explicitly specified. For now, I recommend that you specify the HTTP method for all tools, and I will do so for all `.http` files for my books.

1. Click **Send Request**, and note the response is the same as what was returned by Scalar. A JSON document response containing the first ten products that are in stock and not discontinued, as shown in Visual Studio in *Figure 10.4* and VS Code in *Figure 10.5*:

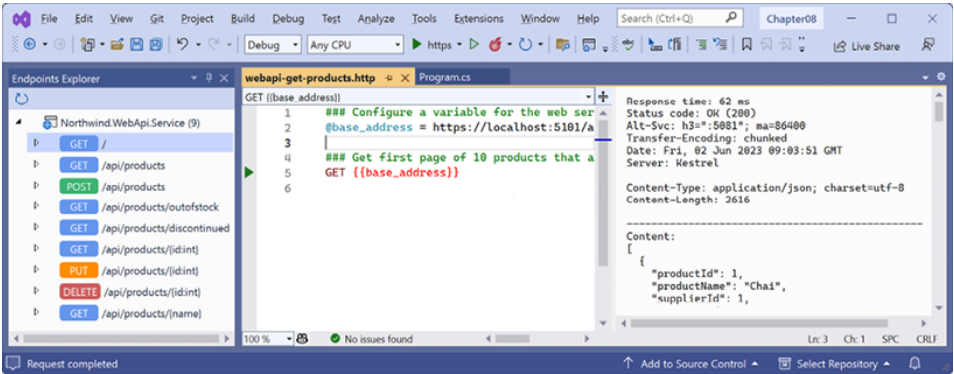


Figure 10.4: Visual Studio getting products from the web service

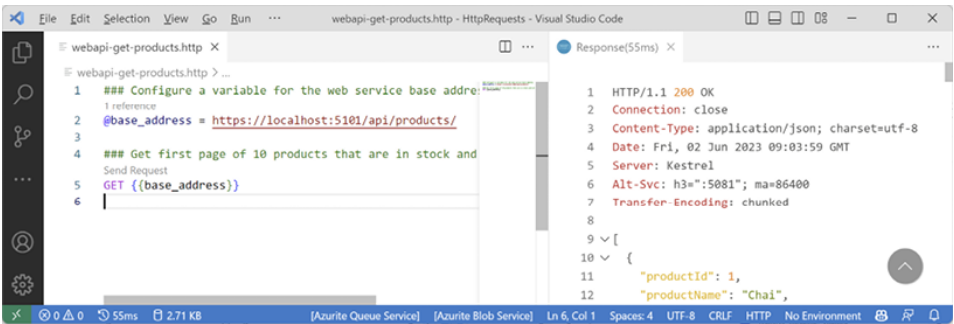


Figure 10.5: VS Code REST Client getting products from the web service

1. In `webapi-get-products.http`, add more requests separated by `###`, as shown in the following file:

```

### Get third page of 10 products that are
in stock and not discontinued GET
{{base_address}}?page=3 ### Get products
that are out-of-stock but not discontinued
GET {{base_address}}outofstock ### Get
products that are discontinued GET
{{base_address}}discontinued ### Get
product 77 GET {{base_address}}77 ### Get
products that contain "man" GET
{{base_address}}man

```

You can execute an HTTP request in Visual Studio by clicking the green triangle “play” button, by right-clicking and selecting **Send Request**, or by pressing *Ctrl + Alt + S*. In VS Code, click **Send Request** above each query, or navigate to **View | Command Palette** and select **Rest Client: Send Request**, or use its keyboard shortcut (*Ctrl + Alt + R* on Windows).

1. In the `HttpRequests` folder, create a file named `webapi-insert-product.http` and modify its contents to contain a **POST** request to insert a new product, as shown in the following code:

```
POST https://localhost:5101/api/products/
Content-Type: application/json {
  "productName": "Harry's Hamburgers",
  "supplierId": 7, "categoryId": 6,
  "quantityPerUnit": "6 per box",
  "unitPrice": 24.99, "unitsInStock": 0,
  "unitsOnOrder": 20, "reorderLevel": 10,
  "discontinued": false }
```

2. Click **Send Request**, and note the response indicates that the new product was added successfully because the status code is `201`, and its location includes its product ID, as shown in *Figure 10.6*:

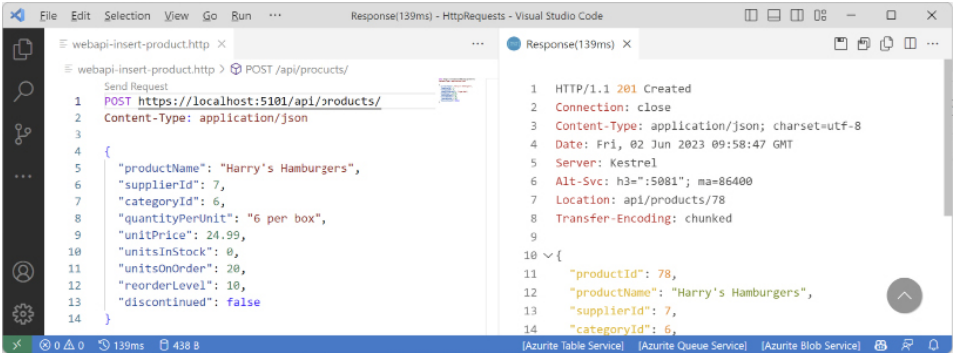


Figure 10.6: REST Client inserting a new product by calling the web API service

Originally, there were 77 products in the Northwind database. The next product ID would be 78. The actual product ID assigned automatically will depend on whether you have previously added any other products, so your assigned number might be higher.

1. In the `HttpRequests` folder, create a file named `webapi-update-product.http` and modify its contents to contain a `PUT` request to update the product with ID 78 (or whatever number was assigned to your Harry's Hamburgers) with a different quantity per unit, unit price, and units in stock, as shown in the following code:

```
PUT https://localhost:5101/api/products/78
Content-Type: application/json {
  "productName": "Harry's Hamburgers",
  "supplierId": 7, "categoryId": 6,
  "quantityPerUnit": "12 per box",
  "unitPrice": 44.99, "unitsInStock": 50,
  "unitsOnOrder": 20, "reorderLevel": 10,
  "discontinued": false }
```

2. Send the request and note that you should get a 204 status code in the response, meaning a successful update.
3. Confirm the product was updated by executing a GET request for the product ID.
4. In the `HttpRequests` folder, create a file named `webapi-delete-product.http` and modify its contents to contain a DELETE request for the new product, as shown in the following code:

```
DELETE https://localhost:5101/api/products/78
```

5. Note the successful response, as shown in *Figure 10.7*:

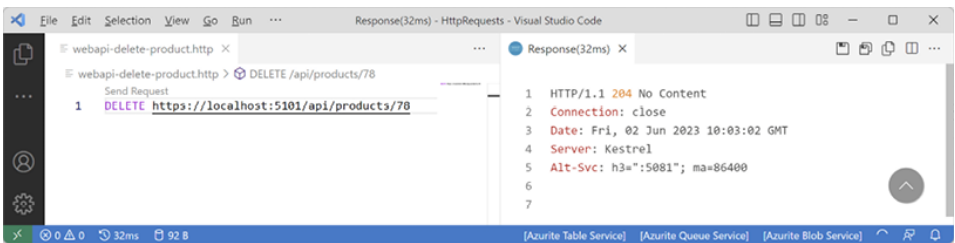


Figure 10.7: Deleting a product using the Web API service

1. Send the request again and note the response contains a 404 status code because the product has now been deleted.
2. Shut down the web server.

Excluding paths from OpenAPI documentation

Sometimes you want to have a path that works but is not shown in the OpenAPI document. Let's see how to remove the service base address that returns a plain text `Hello World!` response from the OpenAPI document:

1. In `WebApplication.Extensions.cs`, for the root path that

returns `Hello World`, exclude it from the OpenAPI document, as shown highlighted in the following code:

```
app.MapGet("/", () => "Hello World!")  
    .ExcludeFromDescription();
```

2. Start the `Northwind.WebApi.Service` project using the `https` profile without debugging and note the path is now not documented.

We have a working web service implemented using ASP.NET Core Minimal API. Now let's attack it! (So that we can learn how to prevent those attacks.) We will start by looking at an often-misunderstood feature of web browsers called CORS.

Relaxing the same origin security policy using CORS

Modern web browsers support multiple tabs so users can visit multiple websites at the same time efficiently. If code executing in one tab could access resources in another tab, then that could be a vector of attack.

All web browsers implement a security feature called the **same origin policy**. This means that only requests that come from the same origin are allowed. For example, if a block of JavaScript is served from the same origin that hosts a web service or served an `<iframe>`, then that JavaScript can call the service and access the data in the `<iframe>`. If a request is made from a different origin, then the request fails. But what counts as the "same origin?"

An origin is defined by:

- **Scheme** aka protocol, for example, `http` or `https`.
- **Port**, for example, `801` or `5101`. The default port for `http` is `80` and for `https` is `443`.
- **Host/domain/subdomain**, for example, `www.example.com`,

`www.example.net`, `example.com`.

If the origin is `https://www.example.com/about-us/`, then the following are *not* the same origin:

- Different scheme: `http://www.example.com/about-us/`
- Different host/domain: `https://www.example.co.uk/about-us/`
- Different subdomain: `https://careers.example.com/about-us/`
- Different port: `https://www.example.com:444/about-us/`

It is the web browser that sets the `Origin` header automatically when making a request. This cannot be overridden.

Warning! The same origin policy does *not* apply to any requests that come from a non-web browser because, in those cases, the programmer could change the `Origin` header anyway. If you create a console app or even an ASP.NET Core project that uses .NET classes like `HttpClient` to make a request, the same origin policy does not apply unless you explicitly set the `Origin` header.

Let's see some examples of calling the web service from a web page with a different origin and from a .NET app.

Configuring HTTP logging for the web service

First, let's enable HTTP logging for the web service and configure it to show the origin of requests:

1. In the `Northwind.WebApi.Service` project, in the `Extensions` folder, add a new class file named

IServiceCollection.Extensions.cs.

2. In `IServiceCollection.Extensions.cs`, import the namespace for controlling which HTTP fields are logged, and then define an extension method for the `IServiceCollection` interface to add HTTP logging, including the `Origin` header and all fields, including the response body, as shown in the following code:

```
using Microsoft.AspNetCore.HttpLogging; //
To use HttpLoggingFields. namespace
Northwind.WebApi.Service.Extensions;
public static class
IServiceCollectionExtensions { public
static IServiceCollection
AddCustomHttpLogging( this
IServiceCollection services) {
services.AddHttpLogging(options => { //
Add the Origin header so it will not be
redacted.
options.RequestHeaders.Add("Origin"); //
By default, the response body is not
included. options.LoggingFields =
HttpLoggingFields.All; }); return
services; } }
```

3. In `Program.cs`, before the call to `builder.Build()`, add a statement to add custom HTTP logging, as shown in the following code:

```
builder.Services.AddCustomHttpLogging();
```

4. In `Program.cs`, before the call to `MapGets()`, add a statement to use HTTP logging, as shown in the following code:

```
app.UseHttpLogging();
```

5. In `appsettings.Development.json`, add an entry to set the level for HTTP logging to `Information`, as shown highlighted in the following configuration:

```
{ "Logging": { "LogLevel": { "Default":  
  "Information", "Microsoft.AspNetCore":  
  "Warning",  
  "Microsoft.AspNetCore.HttpLogging":  
  "Information" } } }
```

Creating a web page JavaScript client

Next, let's create a web page client that will attempt to use JavaScript on a different port to call the web service:

1. Use your preferred code editor to add a new project, as defined in the following list:

- Project template: **ASP.NET Core Web App (Model-View-Controller) / mvc**
- Solution file and folder: `ModernServices`
- Project file and folder:
`Northwind.WebApi.Client.Mvc`
- Other Visual Studio options:
- **Authentication Type: None**
- **Configure for HTTPS: Selected**
- **Enable container support: Cleared**
- **Do not use top-level statements: Cleared**
- **Enlist in Aspire orchestration: Cleared**

11. In the `Northwind.WebApi.Client.Mvc` project, in the

Properties folder, in `launchSettings.json`, change the `applicationUrl` for the `https` profile to use port `5102`, as shown in the following markup:

```
"applicationUrl": "https://  
localhost:5102",
```

12. In the `Northwind.WebApi.Client.Mvc` project file, treat warnings as errors.
13. In the `Views/Home` folder, in `Index.cshtml`, replace the existing markup with the markup below, which has a link to a route that has not been defined yet to define a text box and button, and a JavaScript block that makes a call to the web service to get products that contain a partial name, as shown in the following code:

```
@{ ViewData["Title"] = "Products using  
JavaScript"; } <div class="text-center">  
<h1 class="display-4">@ViewData["Title"]</  
h1> <div> Go to <a href="/home/  
products">Products using .NET</a> </div>  
<div> <input id="productName"  
placeholder="Enter part of a product name"  
> <input id="getProductsButton"  
type="button" value="Get Products" /> </  
div> <div> <table id="productsTable"  
class="table"> <thead> <tr> <th  
scope="col">Product Name</th> </tr> </  
thead> <tbody id="tableBody"> </tbody> </  
table> </div> <script> var baseaddress =  
"https://localhost:5101/"; function  
xhr_load() {  
console.log(this.responseText); var  
products = JSON.parse(this.responseText);
```

```

var out = ""; var i; for (i = 0; i <
products.length; i++) { out += '<tr><td><a
href="' + baseaddress + 'api/products/' +
products[i].productId + '>' +
products[i].productName + '</a></td></
tr>'; }
document.getElementById("tableBody").innerHTML
= out; } function
getProductsButton_click() {
xhr.open("GET", baseaddress + "api/
products/" +
document.getElementById("productName").value);
xhr.send(); }
document.getElementById("getProductsButton")
.addEventListner("click",
getProductsButton_click); var xhr = new
XMLHttpRequest();
xhr.addEventListener("load", xhr_load); </
script> </div>

```

14. Start the Northwind.WebApi.Service project using the https profile without debugging.
15. Start the Northwind.WebApi.Client.Mvc project using the https profile without debugging.

If you are using VS Code, then the web browser will not start automatically. Start Chrome, and then navigate to <https://localhost:5102>.

1. In Chrome, show **Developer Tools** and **Console**.
2. In the **Products using JavaScript** web page, in the text box, enter `man`, click the **Get Products** button, and note the error, as shown in the following output and in *Figure 10.8*:

```
Access to XMLHttpRequest at 'https://localhost:5101/api/products/man' from origin 'https://localhost:5102' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. GET https://localhost:5101/api/products/man net::ERR_FAILED 200
```

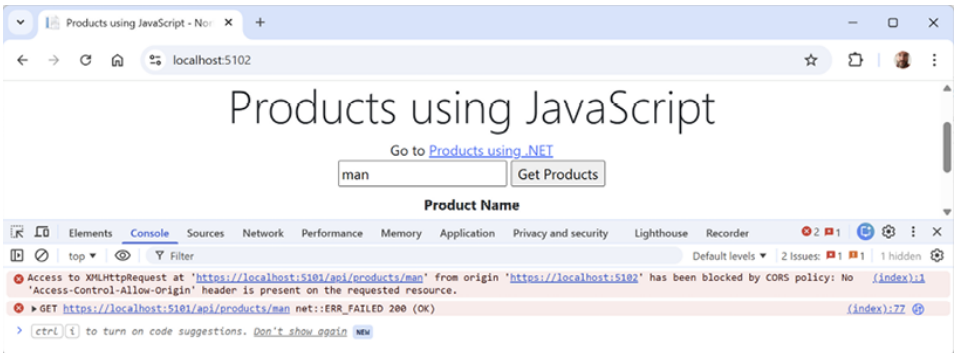


Figure 10.8: CORS error in the Chrome Developer Tools console

1. At the command prompt or terminal for the Northwind.WebApi.Service project, note the HTTP log for the request and that the Host is on a different port number to the Origin so they are not the same origin, as shown highlighted in the following output:

```
info:
Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware:
Request: Protocol: HTTP/2 Method: GET
Scheme: https PathBase: Path: /api/
products/man Accept: */*
Host: localhost:5101
User-Agent: Mozilla/5.0 (Windows NT 10.0;
```

```
Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/113.0.0.0 Safari/537.36
Accept-Encoding: gzip, deflate, br Accept-
Language: en-US,en;q=0.9,sv;q=0.8
```

Origin: <https://localhost:5102>

```
Referer: [Redacted] sec-ch-ua: [Redacted]
sec-ch-ua-mobile: [Redacted] sec-ch-ua-
platform: [Redacted] sec-fetch-site:
[Redacted] sec-fetch-mode: [Redacted] sec-
fetch-dest: [Redacted]
```

2. Also note the output shows that the web service did execute the database query and return the products in a JSON document response to the browser, as shown in the following output:

```
info:
Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware:
Response: StatusCode: 200 Content-Type:
application/json; charset=utf-8 info:
Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware:
ResponseBody:
[{"productId":12,"productName": "Queso
Manchego La
Pastora","supplierId":5,"categoryId":4,
"quantityPerUnit":"10 - 500 g
pkgs.","unitPrice":38.0000,
"unitsInStock":86,"unitsOnOrder":0,"reorderLevel":0,
"discontinued":false,"category":null,"supplier":null},
{"productId":51,"productName":"Manjimup
Dried Apples",
"supplierId":24,"categoryId":7,
"quantityPerUnit":"50 - 300 g
pkgs.","unitPrice":53.0000,
```

```
"unitsInStock":20,"unitsOnOrder":0,"reorderLevel":0,"discontinued":false,"category":null,"supplier":{"name":"Suppliers","url":"http://localhost:12345/Products/Suppliers"},"orderDetails":[]}]
```

Although the browser receives a response containing the data requested, it is the browser that enforces the same origin policy by refusing to reveal the HTTP response to the JavaScript. The web service is not “secured” by CORS.

1. Close the browser(s) and shut down the web servers.

Creating a .NET client

Next, let’s create a .NET client to the web service to see that the same origin policy does not apply to non-web browsers:

1. In the `Northwind.WebApi.Client.Mvc` project, add a reference to the entity models project so that we can use the `Product` class, as shown in the following markup:

```
<ItemGroup> <ProjectReference Include= "..\..\ModernApps\Northwind.EntityModels\Northwind.EntityModels.csproj" /> </ItemGroup>
```

2. Build the `Northwind.WebApi.Client.Mvc` project at the command prompt or terminal by entering the following command:
`dotnet build.`
3. In the `Northwind.WebApi.Client.Mvc` project, in `Program.cs`, import the namespace for working with HTTP headers,

as shown in the following code:

```
using System.Net.Http.Headers; // To use
MediaTyewithQualityHeaderValue.
```

4. In `Program.cs`, before the call to `builder.Build()`, add statements to configure an HTTP client factory to call the web service, as shown in the following code:

```
builder.Services.AddHttpClient(name:
"Northwind.WebApi.Service",
configureClient: options => {
options.BaseAddress = new("https://
localhost:5101/");
options.DefaultRequestHeaders.Accept.Add(
new MediaTypeWithQualityHeaderValue(
"application/json", 1.0)); });
```

5. In the `Controllers` folder, in `HomeController.cs`, import the namespace for the entity models, as shown in the following code:

```
using Northwind.EntityModels; // To use
Product.
```

6. In `HomeController.cs`, add statements to store the registered HTTP client factory in a private readonly field, as shown highlighted in the following code:

```
private readonly IHttpClientFactory
_httpClientFactory; public
HomeController(IHttpClientFactory
```

```
httpClientFactory) { _httpClientFactory =  
httpClientFactory; }
```

7. In `HomeController.cs`, add an asynchronous action method named `Products` that will use the HTTP factory to request products whose name contains a value entered as an optional `name` parameter in a custom MVC route, as shown in the following code:

```
[Route("home/products/{name?}")] public  
async Task<IActionResult> Products(string?  
name) { HttpClient client =  
_httpClientFactory.CreateClient( name:  
"Northwind.WebApi.Service");  
HttpRequestMessage request = new( method:  
HttpMethod.Get, requestUri: $"api/  
products/{name}"); HttpResponseMessage  
response = await  
client.SendAsync(request);  
IEnumerable<Product>? model = await  
response.Content  
.ReadFromJsonAsync<IEnumerable<Product>>();  
ViewData["baseaddress"] =  
client.BaseAddress; return View(model); }
```

8. In the `Views/Home` folder, add a new file named `Products.cshtml`. (The Visual Studio project item template is named **Razor View - Empty**. The Rider project item template is named **Razor MVC View**.)
9. In `Products.cshtml`, modify its contents to output a table of products that match part of a product name entered in a text box, as shown in the following markup:

```

@using Northwind.EntityModels @model
IEnumerable<Product>? @{ ViewData["Title"]
= "Products using .NET"; } <div
class="text-center"> <h1
class="display-4">@ViewData["Title"]</h1>
<div> Go to <a href="/">Products using
JavaScript</a> </div> <form action="/home/
products"> <input name="name"
placeholder="Enter part of a product name"
/> <input type="submit" value="Get
Products" /> </form> <div> <table
class="table"> <thead> <tr> <th
scope="col">Product Name</th> </tr> </
thead> <tbody> @if (Model is not null) {
@foreach (Product p in Model) { <tr><td><a
href="@ (ViewData["baseaddress"])api/
products/ @p.ProductId">p.ProductName</
a></td></tr> } } </tbody> </table> </div>
</div>

```

10. Start the `Northwind.WebApi.Service` project using the `https` profile without debugging.
11. Start the `Northwind.WebApi.Client.Mvc` project using the `https` profile without debugging.
12. On the home page, click the link to go to **Products using .NET**, and note the first ten in-stock, not discontinued products are shown in the table, from **Chai** to **Queso Manchego La Pastora**.
13. In the text box, enter `man`, click **Get Products**, and note that two products are shown in the table, as shown in *Figure 10.9*:

Products using .NET

Go to [Products using JavaScript](#)

Enter part of a product name	Get Products
------------------------------	--------------

Product Name

[Queso Manchego La Pastora](#)

[Manjimup Dried Apples](#)

Figure 10.9: Getting two products from a web service using .NET

It is the .NET HTTP client that is calling the web service, so the same origin policy does not apply. If you were to check the logs at the command line or terminal as you did before, you would see the ports are different, but it does not matter.

1. Click one of the product names to make a direct request to the web service for an individual product and note the response, as shown in the following document:

```
{ "productId":12, "productName": "Queso  
Manchego La Pastora",  
  "supplierId":5, "categoryId":4,  
  "quantityPerUnit": "10 - 500 g  
pkgs.", "unitPrice":38.0000,  
  "unitsInStock":86, "unitsOnOrder":0, "reorderLevel":0,  
  "discontinued":false, "category":null, "supplier":null  
[] }
```

2. Close the browser and shut down the web servers.

Understanding CORS

Cross-Origin Resource Sharing (CORS) is an HTTP-header-based feature

that asks the browser to disable its same-origin security policy in specific scenarios. The HTTP headers indicate which origins should be allowed in addition to the same origin.

Let's enable CORS in the web service so that it can send extra headers to indicate to the browser that it is allowed to access resources from a different origin:

1. In the `Northwind.WebApi.Service` project, in `WebApplication.Extensions.cs`, add an extension method to add CORS support to the web service, as shown in the following code:

```
public static IServiceCollection
AddCustomCors( this IServiceCollection
services) { services.AddCors(options => {
options.AddPolicy(name:
"Northwind.Mvc.Policy", policy => {
policy.WithOrigins("https://
localhost:5102"); }); }); return services;
}
```

2. In `Program.cs`, before calling `builder.Build()`, call the custom extension method to add CORS support, as shown in the following code:

```
builder.Services.AddCustomCors();
```

3. In `Program.cs`, after the call to `UseHttpLogging`, and before mapping the GET requests, add a statement to use the CORS policy, as shown in the following code:

```
app.UseCors(policyName:
"Northwind.Mvc.Policy");
```

4. Start the `Northwind.WebApi.Service` project using the `https` profile without debugging.
5. Start the `Northwind.WebApi.Client.Mvc` project using the `https` profile without debugging.
6. Show **Developer Tools** and the **Console**.
7. On the home page, in the text box, enter `man`, click **Get Products**, and note that the console shows the JSON document returned from the web service, and the table is filled with the two products, as shown in *Figure 10.10*:

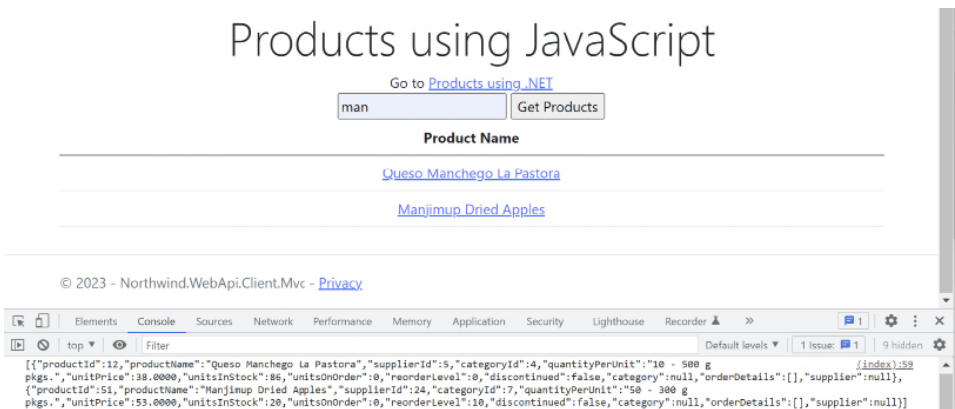


Figure 10.10: A successful cross-origin request to the web service using JavaScript

1. Close the browser and shut down the web server.

Enabling CORS for specific endpoints

In the previous example, we enabled the same CORS policy for the whole web service. You might need finer control at the endpoint level:

1. In the `Northwind.WebApi.Service` project, in `Program.cs`, change the call to `UseCors` to not specify the policy name, as shown highlighted in the following code:

```
// app.UseCors(policyName:
```

```
"Northwind.Mvc.Policy"); // Without a  
named policy the middleware is added but  
not active. app.UseCors();
```

2. In `WebApplication.Extensions.cs`, at the end of the call to `MapGet` that gets products that contain part of a product name, add a call to `RequiresCors`, as shown highlighted in the following code:

```
app.MapGet("api/products/{name}", (  
    [FromServices] NorthwindContext db,  
    [FromRoute] string name) =>  
    db.Products.Where(p =>  
        p.ProductName.Contains(name)))  
    .WithName("GetProductsByName")  
    .WithOpenApi()  
    .Produces<Product[]>(StatusCodes.Status200OK)  
    .RequireCors(policyName:  
        "Northwind.Mvc.Policy");
```

3. Start the `Northwind.WebApi.Service` project using the `https` profile without debugging.
4. Start the `Northwind.WebApi.Client.Mvc` project using the `https` profile without debugging.
5. Show **Developer Tools** and **Console**.
6. On the home page, in the text box, enter `cha`, click **Get Products**, and note that the console shows the JSON document returned from the web service and the table is filled with three products.
7. Close the browser and shut down the web servers.

Understanding other CORS policy options

You can control the:

- Allowed origins, for example, `https://*.example.com/`.
- Allowed HTTP methods, for example, `GET`, `POST`, `DELETE`, and so on.
- Allowed HTTP request headers, for example, `Content-Type`, `Content-Language`, `x-custom-header`, and so on.
- Exposed HTTP response headers, meaning which headers to include unredacted in a response (because, by default, response headers are redacted), for example, `x-custom-header`.

You can learn more about options for CORS policies at the following link: <https://learn.microsoft.com/en-us/aspnet/core/security/cors#cors-policy-options>.

Now that you know that CORS does not secure a web service, let's look at a useful technique that can prevent a common form of attack.

Preventing denial of service attacks using rate limiting

A **denial of service (DoS)** attack is a malicious attempt to disrupt a web service by overwhelming it with requests. If the requests all came from the same place, for example, the same IP address, then it would be relatively easy to cut them off as soon as the attack is detected. But these attacks are often implemented as **distributed DoS (DDoS)** attacks from many locations so you cannot separate attackers from genuine clients.

A different approach is to apply rate limiting to everyone but let through more requests for genuine identified clients.

Genuine clients should only make the minimum requests they need. How many

is reasonable will depend on your service. One way to prevent DDoS attacks would be to limit how many requests are allowed from any client per minute.

This technique is not just useful to prevent attacks. Even genuine clients might accidentally make too many requests, or for a commercial web service, you might want to charge different amounts for different rates, like when controlling a subscription. Commercial web services from Twitter/X to Reddit now charge a *lot* of money for access to their web APIs.

When a client makes requests over a set rate limit, the client should receive either a 429 Too Many Requests or 503 Service Unavailable HTTP response.

Good practice: If you need to build a massively scalable web service and protect its APIs, you should use a cloud service like Azure API Management instead of trying to implement your own rate limiting. You can learn more about this at the following link: <https://learn.microsoft.com/en-us/azure/api-management/>.

Rate limiting using the `AspNetCoreRateLimit` package

`AspNetCoreRateLimit`, a third-party package that targets .NET 6 or later, provides flexible rate-limiting middleware based on the IP address or client ID:

1. In the `Northwind.WebApi.Service` project, add a reference to the `AspNetCoreRateLimit` package, as shown in the following markup:

```
<PackageReference  
  Include="AspNetCoreRateLimit" />
```

2. Build the `Northwind.WebApi.Service` project to restore packages.
3. In `appsettings.Development.json`, add configuration for default rate limit options and client-specific policies, as shown highlighted in the following configuration:

```
{ "Logging": { "LogLevel": { "Default":
"Information", "Microsoft.AspNetCore":
"Warning",
"Microsoft.AspNetCore.HttpLogging":
"Information" } }, "ClientRateLimiting" :{
"EnableEndpointRateLimiting" false:,
"StackBlockedRequests" false:,
"ClientIdHeader" "X-Client-Id":,
"HttpStatusCode" 429:, "EndpointWhitelist"
"get:/api/license" "*/api/status" :[,],
"ClientWhitelist" "dev-id-1" "dev-id-2" :
[,], "GeneralRules" :[ { "Endpoint" "":,
"Period" "10s":, "Limit" 2: }, {
"Endpoint" "":, "Period" "12h":, "Limit"
100: } ] }, "ClientRateLimitPolicies" :{
"ClientRules" :[ { "ClientId" "console-
client-abc123":, "Rules" :[ { "Endpoint"
"":, "Period" "10s":, "Limit" 5: }, {
"Endpoint" "":, "Period" "12h":, "Limit"
250: } ] } ] } }
```

Note:

- `EnableEndpointRateLimiting` is `false`, meaning all endpoints will share the same rules.
- If a client needs to identify itself, it can set a header named `X-Client-Id` to a unique string value.

- If a rate limit is reached for a client, the service will start returning 429 status code responses to that client.
- Two endpoints will be excluded from the global rate limits because they are on the endpoint whitelist. One endpoint is for getting a license, and the other endpoint is for checking the status of the service. We will not actually implement these features and you would want to apply different rate limits to these endpoints; otherwise, someone could call them to bring down your server instead.
- Two client IDs, named `dev-id-1` and `dev-id-2`, will be excluded from the rate limits because they are on the client whitelist. These could be special client accounts for internal developers that are not shared outside the organization.
- Two general (default) rules are configured: the first sets a rate limit of 2 requests every 10 seconds, and the second sets a rate limit of 100 requests every 12 hours.
- Two client-specific rules are configured that are looser than the default rules: for the client ID named `console-client-abc123`, it is allowed to make up to 5 requests every 10 seconds, and up to 250 requests every 12 hours.

1. In `IServiceCollection.Extensions.cs`, import the namespace for working with rate-limiting options, as shown in the following code:

```
using AspNetCoreRateLimit; // To use
ClientRateLimitOptions and so on.
```

2. In `IServiceCollection.Extensions.cs`, define an extension method to load rate-limiting configuration from app settings and set rate-limiting options, as shown in the following code:


```

public static IServiceCollection
AddCustomRateLimiting( this
IServiceCollection services,
ConfigurationManager configuration) { //
Add services to store rate limit counters
and rules in memory.
services.AddMemoryCache();
services.AddInMemoryRateLimiting(); //
Load default rate limit options from
appsettings.json.
services.Configure<ClientRateLimitOptions>(
configuration.GetSection("ClientRateLimiting"));
// Load client-specific policies from
appsettings.json.
services.Configure<ClientRateLimitPolicies>(
configuration.GetSection("ClientRateLimitPolicies"));
// Register the configuration.
services.AddSingleton
<IRateLimitConfiguration,
RateLimitConfiguration>(); return
services; }

```

3. In `Program.cs`, after creating the builder, add statements to load rate-limiting configuration from app settings and set rate-limiting options, as shown in the following code:

```

builder.Services.AddCustomRateLimiting(builder.Configuration);

```

4. In `IServiceCollection.Extensions.cs`, in the call to configure HTTP logging, add a statement to allow two rate-limiting headers to not be redacted, as shown highlighted in the following code:

```

services.AddHttpLogging(options => { //

```

Add the Origin header so it will not be redacted.

`options.RequestHeaders.Add("Origin"); //`
Add the rate limiting headers so they will not be redacted.

`options.RequestHeaders.Add("X-Client-Id");`
`options.ResponseHeaders.Add("Retry-`
`After"); //` *By default, the response body is not included.* `options.LoggingFields =`
`HttpLoggingFields.All; });`

5. In `WebApplication.Extensions.cs`, import the namespace for working with rate-limiting policy stores, as shown in the following code:

```
using AspNetCoreRateLimit; // To use
IClientPolicyStore and so on.
```

6. In `WebApplication.Extensions.cs`, add statements to define an extension method to seed the client policy store, which just means loading the policies from the configuration, and then use client rate limits, as shown in the following code:

```
public static async Task
UseCustomClientRateLimiting(this
WebApplication app) { using (IServiceScope
scope = app.Services.CreateScope()) {
IClientPolicyStore clientPolicyStore =
scope.ServiceProvider
.GetRequiredService<IClientPolicyStore>();
await clientPolicyStore.SeedAsync(); }
app.UseClientRateLimiting(); }
```

7. In `Program.cs`, after calling `UseHttpLogging`, add a call to use client rate limiting, as shown in the following code:

```
await app.UseCustomClientRateLimiting();
```

Creating a rate-limited console client

Now we can create a console app that will be a client to the web service:

1. Use your preferred code editor to add a new console app to the `ModernServices` solution named `Northwind.WebApi.Client.Console`.
2. In the `Northwind.WebApi.Client.Console` project, treat warnings as errors, globally and statically import the `System.Console` class, and add a reference to the entity models project, as shown in the following markup:

```
<ItemGroup> <ProjectReference Include= "..  
  \..\ModernApps\Northwind.EntityModels  
  \Northwind.EntityModels.csproj" /> </  
ItemGroup>
```

3. Build the `Northwind.WebApi.Client.Console` project at the command prompt or terminal to compile the referenced project and copy its assembly to the appropriate `bin` folder.
4. In the `Northwind.WebApi.Client.Console` project, add a new class file named `Program.Helpers.cs`.
5. In `Program.Helpers.cs`, add statements to define a method for the partial `Program` class to write some text in a foreground color, as shown in the following markup:

```
partial class Program { private static
void WriteInColor(string text,
ConsoleColor foregroundColor) {
ConsoleColor previousColor =
ForegroundColor; ForegroundColor =
foregroundColor; Write(text);
ForegroundColor = previousColor; } }
```

6. In `Program.cs`, delete the existing statements. Add statements to prompt the user for a client name to identify it, and then create an HTTP client to make a request to get the first page of products from the web service once per second until the user presses `Ctrl + C` to stop the console app, as shown in the following code:

To save you typing, remember that the full file is available in the GitHub repository:
<https://github.com/markjprice/apps-services-net10/blob/main/code/ModernServices/Northwind.WebApi.Client.Console/Program.cs>.

```
using Northwind.EntityModels; // To use
Product. using System.Net.Http.Json; // To use
ReadFromJsonAsync<T> method. Write("Enter a
client name or press Enter: "); string?
clientName = ReadLine(); if
(string.IsNullOrEmpty(clientName)) {
clientName = $"console-client-
{Guid.NewGuid()}"; } WriteLine($"X-Client-Id
will be: {clientName}"); HttpClient client =
new(); client.BaseAddress = new("https://
localhost:5101");
```

```

client.DefaultRequestHeaders.Accept.Add(new("applicat
json")); // Specify the rate limiting client
id for this console app.
client.DefaultRequestHeaders.Add("X-Client-
Id", clientName); while (true) {
WriteInColor(string.Format("{0:hh:mm:ss}: ",
DateTime.UtcNow), ConsoleColor.DarkGreen); int
waitFor = 1; // Second. try {
HttpResponseMessage response = await
client.GetAsync("api/products"); if
(response.IsSuccessStatusCode) { Product[]?
products = await
response.Content.ReadFromJsonAsync<Product[]>();
if (products != null) { foreach (Product
product in products) {
Write(product.ProductName); Write(", "); }
WriteLine(); } } else {
WriteInColor(string.Format("{0}: {1}",
(int)response.StatusCode, await
response.Content.ReadAsStringAsync()),
ConsoleColor.DarkRed); WriteLine(); } } catch
(Exception ex) { WriteLine(ex.Message); }
await
Task.Delay(TimeSpan.FromSeconds(waitFor)); }

```

Note the following about the preceding code:

- The code prompts the user for a client name and uses a GUID value if they do not supply one.
- A new HTTP client object is instantiated with a base address of the web service and two request headers: an HTTP header to request JSON format for responses and one to specify the client name.
- The client starts an infinite `while` loop. In the loop, it waits for

one second for each iteration and writes the time to the output in dark green.

- In the loop, it makes a request to the web service for products. If the request returns a success status code, it deserializes the response into an array of `Product` instances, and each product name is written to the output. If not successful, it writes the status code and error response as plain text in dark red.

1. If your database server is not running (for example, because you are hosting it in Docker, a virtual machine, or in the cloud), then make sure to start it.
2. Start the `Northwind.WebApi.Service` project using the `https` profile without debugging.
3. Start the `Northwind.WebApi.Client.Console` project without debugging.
4. In the console app, press *Enter* to generate a GUID-based client ID.
5. Start the `Northwind.WebApi.Client.Console` project without debugging again so we have two clients.
6. In the console app, press *Enter* to generate a GUID-based client ID.
7. Note that each client can make two requests before it starts to receive 429 status codes, as shown in the following output and in *Figure 10.11*:

```
Enter a client name or press Enter: X-
Client-Id will be: console-client-
d54c61ba-66bb-4e39-9c1a-7af6e2bf647e
07:32:18: Chai, Chang, Aniseed Syrup, Chef
Anton's Cajun Seasoning, Grandma's
Boysenberry Spread, Uncle Bob's Organic
Dried Pears, Northwoods Cranberry Sauce,
Ikura, Queso Cabrales, Queso Manchego La
Pastora, 07:32:20: Chai, Chang, Aniseed
Syrup, Chef Anton's Cajun Seasoning,
Grandma's Boysenberry Spread, Uncle Bob's
```

Organic Dried Pears, Northwoods Cranberry Sauce, Ikura, Queso Cabrales, Queso Manchego La Pastora, 07:32:21: 429: API calls quota exceeded! maximum admitted 2 per 10s. 07:32:22: 429: API calls quota exceeded! maximum admitted 2 per 10s. 07:32:23: 429: API calls quota exceeded! maximum admitted 2 per 10s. 07:32:24: 429: API calls quota exceeded! maximum admitted 2 per 10s. 07:32:25: 429: API calls quota exceeded! maximum admitted 2 per 10s. 07:32:26: 429: API calls quota exceeded! maximum admitted 2 per 10s. 07:32:27: 429: API calls quota exceeded! maximum admitted 2 per 10s. 07:32:28: Chai, Chang, Aniseed Syrup, Chef Anton's Cajun Seasoning, Grandma's Boysenberry Spread, Uncle Bob's Organic Dried Pears, Northwoods Cranberry Sauce, Ikura, Queso Cabrales, Queso Manchego La Pastora, 07:32:29: Chai, Chang, Aniseed Syrup, Chef Anton's Cajun Seasoning, Grandma's Boysenberry Spread, Uncle Bob's Organic Dried Pears, Northwoods Cranberry Sauce, Ikura, Queso Cabrales, Queso Manchego La Pastora, 07:32:30: 429: API calls quota exceeded! maximum admitted 2 per 10s. 07:32:31: 429: API calls quota exceeded! maximum admitted 2 per 10s. 07:32:32: 429: API calls quota exceeded! maximum admitted 2 per 10s.

```
C:\apps-services-net8\Chapter08\Northwind.WebApi.Client.Console\bin\Debug\net8.0\Northwind.WebApi.Client.Console.exe X + - □ X
Enter a client name or press Enter:
X-Client-Id will be: console-client-8aeb2e7e-3679-4fba-ba4b-93dc2fa7429f
12:48:40: Chai, Chang, Aniseed Syrup, Chef Anton's Cajun Seasoning, Grandma's Boysenberry Spread, Uncle Bob's
Organic Dried Pears, Northwoods Cranberry Sauce, Ikura, Queso Cabrales, Queso Manchego La Pastora,
12:48:42: Chai, Chang, Aniseed Syrup, Chef Anton's Cajun Seasoning, Grandma's Boysenberry Spread, Uncle Bob's
Organic Dried Pears, Northwoods Cranberry Sauce, Ikura, Queso Cabrales, Queso Manchego La Pastora,
12:48:43: 429: API calls quota exceeded! maximum admitted 2 per 10s.
12:48:44: 429: API calls quota exceeded! maximum admitted 2 per 10s.
12:48:45: 429: API calls quota exceeded! maximum admitted 2 per 10s.
12:48:46: 429: API calls quota exceeded! maximum admitted 2 per 10s.
12:48:47: 429: API calls quota exceeded! maximum admitted 2 per 10s.
12:48:48: 429: API calls quota exceeded! maximum admitted 2 per 10s.
12:48:49: 429: API calls quota exceeded! maximum admitted 2 per 10s.
12:48:50: 429: API calls quota exceeded! maximum admitted 2 per 10s.
12:48:51: Chai, Chang, Aniseed Syrup, Chef Anton's Cajun Seasoning, Grandma's Boysenberry Spread, Uncle Bob's
Organic Dried Pears, Northwoods Cranberry Sauce, Ikura, Queso Cabrales, Queso Manchego La Pastora,
12:48:52: Chai, Chang, Aniseed Syrup, Chef Anton's Cajun Seasoning, Grandma's Boysenberry Spread, Uncle Bob's
Organic Dried Pears, Northwoods Cranberry Sauce, Ikura, Queso Cabrales, Queso Manchego La Pastora,
12:48:53: 429: API calls quota exceeded! maximum admitted 2 per 10s.
```

Figure 10.11: A console app exceeding its web service rate limit

1. Stop the two console apps. Leave the web service running.
2. In the command line for the web service, note the HTTP logs that show each request from the console client with its client ID sent as a header named `X-Client-Id`, the request being blocked because that client has exceeded its quota, and a response that contains a header named `Retry-After` showing the number of seconds the client should wait before retrying, as shown highlighted in the following output:

```
info:
Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware:
Request: Protocol: HTTP/1.1 Method: GET
Scheme: https PathBase: Path: /api/
products Accept: application/json Host:
localhost:5101
X-Client-Id: console-client-
d54c61ba-66bb-4e39-9c1a-7af6e2bf647e
info:
AspNetCoreRateLimit.ClientRateLimitMiddleware:
Request get:/api/products from ClientId
console-client-
d54c61ba-66bb-4e39-9c1a-7af6e2bf647e has
been blocked, quota 2/10s exceeded by 3.
```



```
Blocked by rule *, TraceIdentifier
0HMIKGNJQEK5P:0000000E. MonitorMode: False
info:
Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware
Response: StatusCode: 429 Content-Type:
text/plain
Retry-After: 6
info:
Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware
ResponseBody: API calls quota exceeded!
maximum admitted 2 per 10s.
```

3. In the `Northwind.WebApi.Client.Console` project, in `Program.cs`, before writing the error message to the console in dark red, add statements to read the `Retry-After` header to get the number of seconds to wait for, as shown highlighted in the following code:

```
string retryAfter = response.Headers
    .GetValues("Retry-After").ToArray()[0]; if
(int.TryParse(retryAfter, out waitFor)) {
    retryAfter = string.Format("I will retry
    after {0} seconds.", waitFor); }
WriteInColor(string.Format("{0}: {1} {2}",
    (int)response.StatusCode, await
    response.Content.ReadAsStringAsync(),
    retryAfter), ConsoleColor.DarkRed);
```

Note the `waitFor` variable is set from the `Retry-After` header value. This is later used to pause the console app using an asynchronous delay, as shown in the following code:

```
await
```

```
Task.Delay(TimeSpan.FromSeconds
```

1. Start the `Northwind.WebApi.Client.Console` project without debugging.
2. In the console app, press *Enter* to generate a GUID-based client ID.
3. Note the console app will now sensibly wait for the suggested number of seconds before making its next call to the service, as shown in the following output:

```
Enter a client name: X-Client-Id will be:
console-client-
add7613f-51a9-4c4a-8ec7-0244203d2e19
07:45:01: Chai, Chang, Aniseed Syrup, Chef
Anton's Cajun Seasoning, Grandma's
Boysenberry Spread, Uncle Bob's Organic
Dried Pears, Northwoods Cranberry Sauce,
Ikura, Queso Cabrales, Queso Manchego La
Pastora, 07:45:02: Chai, Chang, Aniseed
Syrup, Chef Anton's Cajun Seasoning,
Grandma's Boysenberry Spread, Uncle Bob's
Organic Dried Pears, Northwoods Cranberry
Sauce, Ikura, Queso Cabrales, Queso
Manchego La Pastora, 07:45:03: 429: API
calls quota exceeded! maximum admitted 2
per 10s. I will retry after 8 seconds.
07:45:11: Chai, Chang, Aniseed Syrup, Chef
Anton's Cajun Seasoning, Grandma's
Boysenberry Spread, Uncle Bob's Organic
Dried Pears, Northwoods Cranberry Sauce,
Ikura, Queso Cabrales, Queso Manchego La
Pastora, 07:45:12: Chai, Chang, Aniseed
```

```
Syrup, Chef Anton's Cajun Seasoning,  
Grandma's Boysenberry Spread, Uncle Bob's  
Organic Dried Pears, Northwoods Cranberry  
Sauce, Ikura, Queso Cabrales, Queso  
Manchego La Pastora, 07:45:13: 429: API  
calls quota exceeded! maximum admitted 2  
per 10s. I will retry after 8 seconds.
```

4. Stop and restart the `Northwind.WebApi.Client.Console` project without debugging.
5. In the console app, enter the name `dev-id-1`, and note that the rate limit does not apply to this console app client. This could be a special account for internal developers.
6. Stop and restart the `Northwind.WebApi.Client.Console` project without debugging.
7. In the console app, enter the name `console-client-abc123`, and note that the rate limit is different for this console app client ID, as shown in the following output:

```
info:  
AspNetCoreRateLimit.ClientRateLimitMiddleware 01  
Request get:/api/products from ClientId  
console-client-abc123 has been blocked,  
quota 5/10s exceeded by 1. Blocked by rule  
*, TraceIdentifier 0HMIKGS1TPSHJ:00000006.  
MonitorMode: False
```

Rate limiting using ASP.NET Core middleware

ASP.NET Core 7 introduced its own basic rate-limiting middleware, initially

distributed as a separate NuGet package but now included with ASP.NET Core. It has a dependency on another Microsoft package, `System.Threading.RateLimiting`. It is not as feature-rich as the third-party package and we will not cover it in this book, although I have written an online-only section at the following link:

<https://github.com/markjprice/apps-services-net10/blob/main/docs/ch10-rate-limiting.md>

You can learn about the ASP.NET Core rate limiter at the following link: <https://learn.microsoft.com/en-us/aspnet/core/performance/rate-limit>.

Protecting your web services from attacks is important. What about improving the performance of your web service? Is there anything we can do about that?

Improving startup time and resources using native AOT

Native AOT produces apps and services that are:

- **Self-contained**, meaning they can run on systems that do not have the .NET runtime installed.
- **Ahead-of-time (AOT) compiled to native code**, meaning a faster startup time and a potentially smaller memory footprint. This can have a positive impact when you have lots of instances (for example, when deploying massively scalable microservices) that are frequently stopped and restarted.

Native AOT compiles **intermediate code (IL)** to native code at the time of publishing, rather than at runtime using the **Just-In-Time (JIT)** compiler. But native AOT apps and services must target a specific runtime environment like Windows x64 or Linux ARM.

Since native AOT happens at publish time, while debugging and working live on a project in your code editor, it uses the runtime JIT compiler, not native AOT, even if you have AOT enabled in the project! But some features that are incompatible with native AOT will be disabled or throw exceptions, and a source analyzer is enabled to show warnings about potential code incompatibilities.

Limitations of native AOT

Native AOT has limitations, some of which are shown in the following list:

- No dynamic loading of assemblies.
- No runtime code generation, for example, using `System.Reflection.Emit`.
- It requires trimming, which has its own limitations.
- The assembly must be self-contained, so they must embed any libraries they call, which increases their size.

Although your own apps and services might not use the features listed above, major parts of .NET itself do. For example, ASP.NET Core MVC (including Web API services that use controllers) and EF Core perform runtime code generation to implement their functionality.

The .NET teams are hard at work making as much of .NET compatible with native AOT as possible, as soon as possible. But .NET 8 only includes basic support for ASP.NET Core if you use Minimal API, and no support for EF Core.

My guess is that .NET 9 will include support for ASP.NET Core MVC and some parts of EF Core, but it could take until .NET 10 before we can all confidently use most of .NET and know we can build our apps and services with native AOT to gain the benefits.

The native AOT publishing process includes code analyzers to warn you if you use any features that are not supported, but not all packages have been annotated to work well with this yet.

The most common annotation used to indicate that a type or member does not support AOT is the `[RequiresDynamicCode]` attribute.

You can learn more about AOT warnings at the following link: <https://learn.microsoft.com/en-us/dotnet/core/deploying/native-aot/fixing-warnings>.

Reflection and native AOT

Reflection is a runtime capability that lets a program inspect and interact with its own code. Reflection is frequently used for runtime inspection of type metadata, dynamic invocation of members, and code generation.

Native AOT does allow some reflection features, but the trimming performed during the native AOT compilation process cannot statically determine when a type has members that might be only accessed via reflection. These members would be removed by AOT, which would then cause a runtime exception.

Good practice: Developers must annotate their types with `[DynamicallyAccessedMembers]` to indicate a member that is only dynamically accessed via reflection and should therefore be left untrimmed.

Native AOT for ASP.NET Core

.NET 7 only supported native AOT with console apps and class libraries on Windows or Linux. It did not support macOS or ASP.NET Core. .NET 8 is the first version to support macOS and parts of ASP.NET Core.

The following ASP.NET Core features are fully supported: gRPC, CORS,

HealthChecks, HttpLogging, Localization, OutputCaching, RateLimiting, RequestDecompression, ResponseCaching, ResponseCompression, Rewrite, StaticFiles, and WebSockets.

The following ASP.NET Core features are partially supported: Minimal API.

The following ASP.NET Core features are not supported (yet): MVC, Blazor Server, SignalR, Authentication (except JWT), Session, and SPA.

As you have previously seen, you implement an ASP.NET Core Minimal API web service by mapping an HTTP request to a lambda expression, for example, as shown in the following code:

```
app.MapGet("/", () => "Hello World!");
```

At runtime, ASP.NET Core uses the `RequestDelegateFactory` (**RDF**) class to convert your `MapX` calls into `RequestDelegate` instances. But this is dynamic code, so it is not compatible with native AOT.

In ASP.NET Core 8, when native AOT is enabled, the runtime use of RDF is replaced with a source generator named **Request Delegate Generator (RDG)** that performs similar work but at compile time. This makes sure the code generated is statically analyzable by the native AOT publish process.

Requirements for native AOT

There are additional requirements for different operating systems:

- On Windows, you must install the Visual Studio **Desktop development with C++** workload with all default components.
- On Linux, you must install the compiler toolchain and developer packages for libraries that the .NET runtime depends on. For example, for Ubuntu 18.04 or later: `sudo apt-get install clang zlib1g-dev`.

Warning! Cross-platform native AOT publishing is not supported. This means that you must run the publish on the operating system that you will deploy to. For example, you cannot publish a native AOT project on Linux to later run on Windows.

Enabling native AOT for a project

To enable native AOT publishing in a project, add the `<PublishAot>` element to the project file, as shown highlighted in the following markup:

```
<PropertyGroup> <TargetFramework>net10.0</  
TargetFramework> <PublishAot>true</PublishAot>
```

Enabling JSON serialization with native AOT

JSON serialization with native AOT requires the use of the `System.Text.Json` source generator. All model types passed as parameters or return values must be registered with a `JsonSerializerContext`, as shown in the following code:

```
[JsonSerializable(typeof(Product))] // A single  
Product. [JsonSerializable(typeof(Product[]))]  
// An array of Products. public partial class  
MyJsonSerializerContext :  
JsonSerializerContext { }
```

Your custom JSON serializer context must be added to the service dependencies, as shown in the following code:


```
builder.Services.ConfigureHttpJsonOptions(options
=> {
options.SerializerOptions.AddContext<MyJsonSerializer
});
```

Building a native AOT project

Now let's see a practical example using the new project template:

1. In the solution named `ModernServices`, add a native AOT-compatible web service project, as defined in the following list:
 - Project template: **ASP.NET Core Web API (native AOT)** / `webapiaot`

This is a new project template introduced with .NET 8. It is different from the **Web API** / `webapi` project template. It does not have an option to use controllers since native AOT support is currently Minimal API-only. It also does not have an option for HTTPS because HTTPS is often handled by a reverse-proxy in cloud-native deployments. In Rider, select **ASP.NET Core Web Application** and then select the **Type of Web API (native AOT)**.

- Solution file and folder: `ModernServices`

- Project file and folder:
Northwind.MinimalAot.Service
- **Enable container support:** Cleared
- **Do not use top-level statements:** Cleared
- **Enable OpenAPI support:** Cleared
- **Enlist in Aspire orchestration:** Cleared

9. In the Properties folder, in `launchSettings.json`, note only `http` is configured; delete the `launchUrl` and modify the port to use 5103, as shown highlighted in the following configuration:

```
{ "$schema": "http://json.schemastore.org/
launchsettings.json", "profiles": {
"http": { "commandName": "Project",
"dotnetRunMessages": true,
"launchBrowser": true, "launchUrl": "",
"applicationUrl": "http://localhost:5103",
"environmentVariables": {
"ASPNETCORE_ENVIRONMENT": "Development" }
} } }
```

10. In the project file, change invariant globalization to `false`, note that native AOT publishing is enabled, and add a package reference for SQL Server for ADO.NET, as shown highlighted in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
<PropertyGroup> <TargetFramework>net10.0</
TargetFramework> <Nullable>enable</
Nullable> <ImplicitUsings>enable</
ImplicitUsings>
<InvariantGlobalization>>false</
InvariantGlobalization> <PublishAot>true</
PublishAot> </PropertyGroup> <ItemGroup>
```

```
<PackageReference  
Include="Microsoft.Data.SqlClient" /> </  
ItemGroup> </Project>
```

11. Add a new class file named `Product.cs`, and modify its contents to define a class that represents just the three columns we want from each row in the `Products` table, as shown in the following code:

```
namespace Northwind.Models; public class  
Product { public int ProductId { get; set;  
} public string? ProductName { get; set; }  
public decimal? UnitPrice { get; set; } }
```

12. Add a new class file named `NorthwindJsonSerializerContext.cs`.
13. In `NorthwindJsonSerializerContext.cs`, define a class that enables a `Product` and a list of `Product` objects to be serialized as JSON, as shown in the following code:

```
using System.Text.Json.Serialization; //  
To use JsonSerializerContext. using  
Northwind.Models; // To use Product.  
namespace Northwind.Serialization;  
[JsonSerializable(typeof(Product))]  
[JsonSerializable(typeof(List<Product>))]  
internal partial class  
NorthwindJsonSerializerContext :  
JsonSerializerContext { }
```

14. Delete the `Northwind.MinimalAot.Service.http` file.
15. Add a new class file named `WebApplication.Extensions.cs`.
16. In `WebApplication.Extensions.cs`, define an extension

method for the `WebApplication` class that maps some HTTP GET requests to return a plain text response, and a list of either all products or products with a minimum price from the Northwind database using ADO.NET, as shown in the following code:

```
using Microsoft.Data.SqlClient; // To use
SqlConnection and so on. using
Northwind.Models; // To use Product. using
System.Data; // To use CommandType.
namespace Packt.Extensions; public static
class WebApplicationExtensions { public
static WebApplication MapGets(this
WebApplication app) { //
app.MapGet(pattern, handler);
app.MapGet("/", () => "Hello from a native
AOT minimal API web service.");
app.MapGet("/products", GetProducts);
app.MapGet("/products/
{minimumUnitPrice:decimal?}",
GetProducts); return app; } private static
List<Product> GetProducts(decimal?
minimumUnitPrice = null) {
SqlConnectionStringBuilder builder =
new(); builder.InitialCatalog =
"Northwind";
builder.MultipleActiveResultSets = true;
builder.Encrypt = true;
builder.TrustServerCertificate = true;
builder.ConnectTimeout = 10; // Default is
30 seconds. builder.DataSource =
"127.0.0.1,443"; // SQL Server in a
container. builder.UserID =
Environment.GetEnvironmentVariable("MY_SQL_USR");
builder.Password =
Environment.GetEnvironmentVariable("MY_SQL_PWD");
```

```

builder.PersistSecurityInfo = false; using
SqlConnection connection =
new(builder.ConnectionString);
connection.Open(); SqlCommand cmd =
connection.CreateCommand();
cmd.CommandType = CommandType.Text;
cmd.CommandText = "SELECT ProductId,
ProductName, UnitPrice FROM Products"; if
(minimumUnitPrice.HasValue) {
cmd.CommandText += " WHERE UnitPrice >=
@minimumUnitPrice";
cmd.Parameters.AddWithValue("minimumUnitPrice",
minimumUnitPrice); } using SqlDataReader r
= cmd.ExecuteReader(); List<Product>
products = new(); while (r.Read()) {
Product p = new() { ProductId =
r.GetInt32("ProductId"), ProductName =
r.GetString("ProductName"), UnitPrice =
r.GetDecimal("UnitPrice") };
products.Add(p); } r.Close(); return
products; } }

```

With native AOT, we cannot use EF Core, so we are using the lower-level ADO.NET SqlClient API. This is faster and more efficient anyway. In the future, perhaps with .NET 11 or .NET 12, we will be able to use our EF Core model for Northwind instead.

1. In `Program.cs`, note the call to the `CreateSlimBuilder` method, which ensures that only the essential features of ASP.NET Core are enabled by default, so it minimizes the deployed web service size, as shown in the following code:

```
var builder =  
WebApplication.CreateSlimBuilder(args);
```

The `CreateSlimBuilder` method does not include support for HTTPS or HTTP/3, although you can add those back in yourself if you need them. It does support JSON file configuration for `appsettings.json` and logging.

1. In `Program.cs`, after the call to `builder.Build()`, delete the statements that generated some sample todos and map some endpoints, as shown in the following code:

```
var sampleTodos = new Todo[] { new(1,  
    "Walk the dog"), new(2, "Do the dishes",  
    DateOnly.FromDateTime(DateTime.Now)),  
    new(3, "Do the laundry",  
    DateOnly.FromDateTime(DateTime.Now.AddDays(1))),  
    new(4, "Clean the bathroom"), new(5,  
    "Clean the car",  
    DateOnly.FromDateTime(DateTime.Now.AddDays(2)))  
}; var todosApi = app.MapGroup("/todos");  
todosApi.MapGet("/", () => sampleTodos);  
todosApi.MapGet("/{id}", (int id) =>  
    sampleTodos.FirstOrDefault(a => a.Id ==  
    id) is { } todo ? Results.Ok(todo) :  
    Results.NotFound());
```

2. In `Program.cs`, at the bottom of the file, delete the statements that define the `Todo` record and `AppJsonSerializerContext` class, as shown in the following code:

```
public record Todo(int Id, string? Title,
DateOnly? DueBy = null, bool IsComplete =
false); [JsonSerializable(typeof(Todo[]))]
internal partial class
AppJsonSerializerContext :
JsonSerializerContext { }
```

3. In `Program.cs`, modify the statement that inserts the JSON serialization context to use the Northwind one, and then before running the web app, call the `MapGets` method, as shown highlighted in the following code:

```
var builder =
WebApplication.CreateSlimBuilder(args);
builder.Services.ConfigureHttpJsonOptions(options
=> {
options.SerializerOptions.TypeInfoResolverChain
.Insert(0,
NorthwindJsonSerializerContext.Default);
}); var app = builder.Build();
app.MapGets(); app.Run();
```

4. If your database server is not running (for example, because you are hosting it in Docker, a virtual machine, or in the cloud), then make sure to start it.
5. Start the web service project using the `http` profile:
 - If you are using Visual Studio, then select the **http** profile in the drop-down list and then navigate to **Debug | Start Without Debugging** or press `Ctrl + F5`.
 - If you are using VS Code, then enter the command `dotnet run --launch-profile http`, manually start a web browser, and navigate to the web

service: `http://localhost:5103/`.

Warning! By default for an AOT web service only HTTP is enabled. If you try to navigate to `https://localhost:5103/` it will fail.

1. In the web browser, note the plain text response: Hello from a native AOT minimal API web service.
2. In the address bar, append `/products`, and note the array of products in the response, as shown in the following partial output:

```
[{"productId":1,"productName":"Chai","unitPrice":2.50},
{"productId":2,"productName":"Chang","unitPrice":4.99},
{"productId":3,"productName":"Aniseed Syrup","unitPrice":10.0000},
{"productId":4,"productName":"Chef Anton's Cajun Seasoning", "unitPrice":22.0000},
{"productId":5,"productName":"Chef Anton's Gumbo Mix", "unitPrice":21.3500},
{"productId":6,"productName":"Grandma's Boysenberry Spread", "unitPrice":25.0000},
{"productId":7,"productName":"Uncle Bob's Organic Dried Pears",
"unitPrice":30.0000},
{"productId":8,"productName":"Northwoods Cranberry Sauce", "unitPrice":40.0000},
{"productId":9,"productName":"Mishi Kobe Niku","unitPrice":97.0000},
{"productId":10,"productName":"Ikura","unitPrice":14.99},
{"productId":11,"productName":"Queso Cabrales","unitPrice":21.0000},
{"productId":12,"productName":"Queso Manchego La Pastora",
```



```
"unitPrice":38.0000},
```

3. In the address bar, append `/products/100`, and note the array of two products in the response, as shown in the following partial output:

```
[{"productId":29,"productName":"Thüringer  
Rostbratwurst","unitPrice":123.7900},  
{"productId":38,"productName":"Côte de  
Blaye","unitPrice":263.5000}]
```

4. Close the browser and shut down the web server.

Publishing a native AOT project

A service that functions correctly during development when the service is untrimmed and JIT-compiled could still fail once you publish it using native AOT. You should therefore perform a publish before assuming your project will work.

If your project does not produce any AOT warnings at publish time, you can then be confident that your service will work after publishing for AOT.

Let's review the source-generated code and publish our web service:

1. In the `Northwind.MinimalAot.Service` project file, add an element to emit compiler-generated files, as shown highlighted in the following markup:

```
<PropertyGroup> <TargetFramework>net10.0</  
TargetFramework> ...  
<EmitCompilerGeneratedFiles>true</  
EmitCompilerGeneratedFiles> </  
PropertyGroup>
```

2. Build the `Northwind.MinimalAot.Service` project.
3. If you are using Visual Studio, toggle **Show All Files** in **Solution Explorer**.
4. Expand the `obj\Debug\net10.0\generated` folder, and note the folders and files that have been created by the source generators for AOT and JSON serialization, as well as that you will be opening some of these files in the next few steps, as shown in *Figure 10.12*:

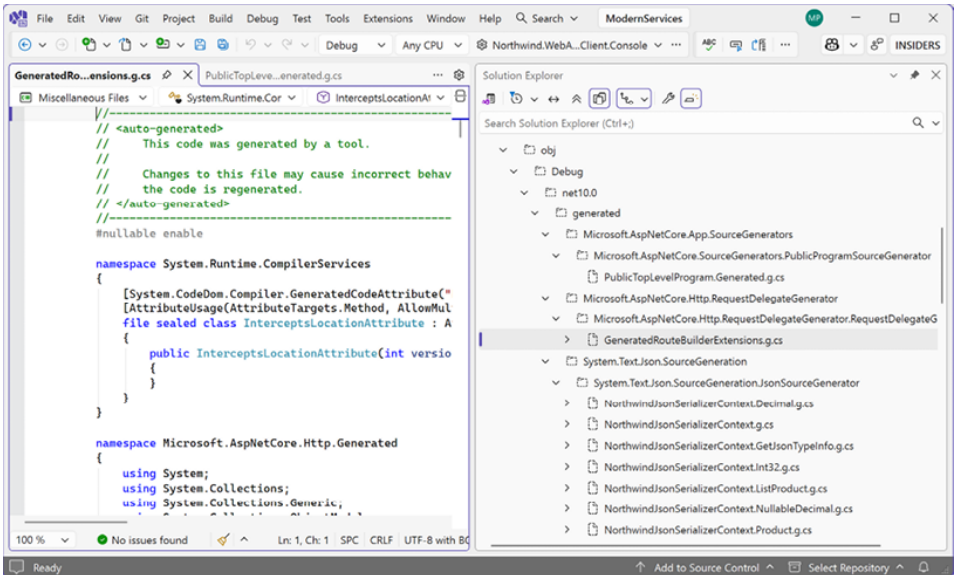


Figure 10.12: Folders and files created by source generators in an AOT web service project

1. Open the `GeneratedRouteBuilderExtensions.g.cs` file, and note it contains code to define the mapped routes for the minimal API web service.
2. Open the `NorthwindJsonSerializerContext.Decimal.g.cs` file, and note it contains code to serialize a `decimal` value passed as the minimum unit price parameter to one of the routes.
3. Open the `NorthwindJsonSerializerContext.ListProduct.g.cs`

file, and note it contains code to serialize a list of `Product` objects returned as a response for two of the routes.

4. In the `Northwind.MinimalAot.Service` folder, at the command prompt or terminal, publish the web service using native AOT, as shown in the following command:

```
dotnet publish
```

5. Note the message about generating native code and trim warnings for packages like `Microsoft.Data.SqlClient`, as shown in the following partial output:

```
Restore complete (0.7s)
Northwind.MinimalAot.Service net10.0 win-
x64 succeeded with 3 warning(s) (53.1s) →
bin\Release\net10.0\win-x64\publish\ C:
\Users\markj\.nuget\packages
\microsoft.data.sqlclient\6.1.3\runtimes
\win\lib
\net9.0\Microsoft.Data.SqlClient.dll :
warning IL2104: Assembly
'Microsoft.Data.SqlClient' produced trim
warnings. For more information see
https://aka.ms/il2104 C:\Users\markj
\nuget\packages\microsoft.data.sqlclient
\6.1.3\runtimes\win\lib
\net9.0\Microsoft.Data.SqlClient.dll :
warning IL3053: Assembly
'Microsoft.Data.SqlClient' produced AOT
analysis warnings. C:\Users\markj\nuget
\packages
\system.configuration.configurationmanager
\9.0.4\lib
```

```
\net9.0\System.Configuration.ConfigurationManager
: warning IL2104: Assembly
'System.Configuration.ConfigurationManager'
produced trim warnings. For more
information see https://aka.ms/il2104
Build succeeded with 3 warning(s) in 54.2s
```

We did not call any potentially trimmed members, so we can ignore the preceding warning.

1. Start **File Explorer** and open the `bin\Release\net10.0\win-x64\publish` folder and note the EXE file is about 30 MB. This and the `Microsoft.Data.SqlClient.SNI.dll` file are the only files that need to be deployed onto another Windows computer for the web service to work. The `appsettings.json` files are only needed to override configuration if needed. The PDB file is only needed if debugging.
2. Run the `Northwind.MinimalAot.Service.exe`, and note the web service starts up very fast and it will use port 5000 by default, as shown in *Figure 10.13*:

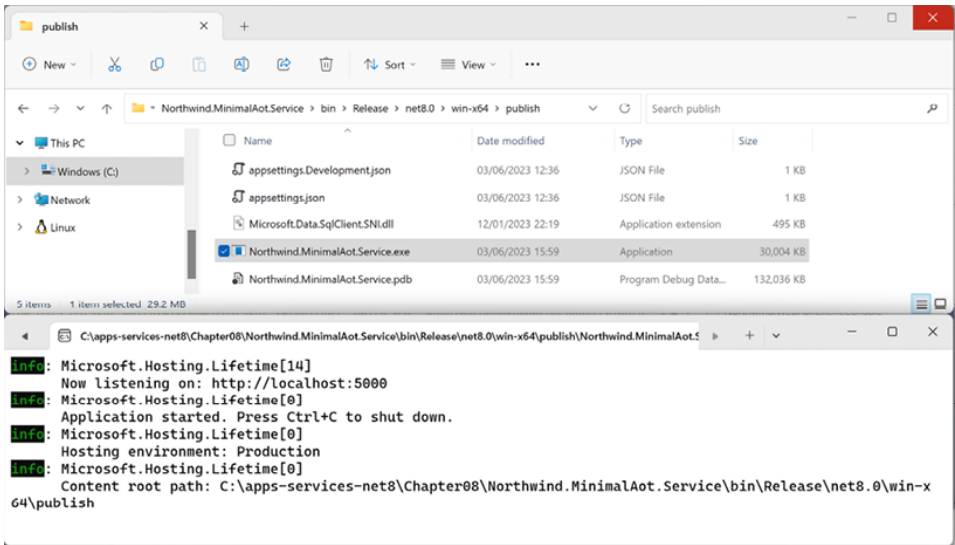


Figure 10.13: File Explorer showing published executable and running the AOT web service

1. Start a web browser, navigate to `http://localhost:5000/`, and note the web service works correctly by returning the plain text response.
2. Navigate to `http://localhost:5000/products/100`, and note the web service responds with the two products that have a minimum unit price of 100.
3. Close the web browser and shut down the web service.
4. In the `Northwind.WebApi.Service` project file, at the command prompt or terminal, publish the web service, as shown in the following command:

```
dotnet publish
```

5. Start **File Explorer**, open the `bin\Release\net10.0\win-x64\publish` folder, and note the `Northwind.WebApi.Service.exe` file is less than 154 KB. That is because it is framework-dependent, meaning it needs .NET installed on the computer to work. Also, there are many files that must be deployed

along with the EXE file that add up to about 14 MB.

6. Run the `Northwind.MinimalAot.Service.exe` and note the web service starts up slower than the AOT version, and that it uses port 5000 by default.
7. Start a web browser, navigate to `http://localhost:5000/`, and note the web service works correctly by returning the plain text response.
8. Navigate to `http://localhost:5000/products/100`, and note the web service responds with the two products that have a minimum unit price of 100, but the response is slower than the AOT version.
9. Close the web browser and shut down the web service.

Many .NET developers have been waiting for AOT compilation for a long time. Microsoft has started delivering on the promise, and it will expand to cover more project types over the next few major versions, so it's a technology to keep an eye on.

Let's end this chapter by looking at identity services and how you can use JWT bearer authentication to protect your web services.

Understanding identity services

Identity services are used to authenticate and authorize users. It is important for these services to implement open standards so that you can integrate disparate systems. Common standards include **OpenID Connect** and **OAuth 2.0**.

Microsoft has no plans to officially support third-party authentication servers like **IdentityServer4** because “creating and sustaining an authentication server is a full-time endeavor, and Microsoft already has a team and a product in that area, Azure Active Directory, which allows 500,000 objects for free.”

JWT bearer authorization

JSON Web Token (JWT) is a standard that defines a compact and secure method to transmit information as a JSON object. The JSON object is digitally

signed, so it can be trusted. The most common scenario for using JWT is authorization.

A user logs in to a trusted party using credentials like a username and password, a biometric scan, or two-factor authentication, and the trusted party issues a JWT. This is then sent with every request to the secure web service.

In their compact form, JWTs consist of three parts separated by dots. These parts are the *header*, *payload*, and *signature*, as shown in the following format: `aaa.bbb.ccc`. The header and payload are Base64 encoded.

Authenticating service clients using JWT bearer authentication

During local development, the `dotnet user-jwts` command-line tool is used to create and manage local JWTs. The values are stored in a JSON file in the local machine's user profile folder.

Let's secure the web service using JWT bearer authentication and test it with a local token:

1. In the `Northwind.WebApi.Service` project, add a reference to the package for JWT bearer authentication, as shown in the following markup:

```
<PackageReference  
  Include="Microsoft.AspNetCore.Authentication.JwtB  
  />
```

2. Build the `Northwind.WebApi.Service` project to restore packages.
3. In `Program.cs`, after creating the `builder`, add statements to add authorization and authentication using JWT, as shown highlighted in the following code:

```
var builder =
    WebApplication.CreateBuilder(args);
builder.Services.AddAuthorization();
builder.Services.AddAuthentication(defaultScheme:
"Bearer") .AddJwtBearer();
```

4. In `Program.cs`, after building the app, add a statement to use authorization, as shown highlighted in the following code:

```
var app = builder.Build();
app.UseAuthorization();
```

5. In the `Extensions` folder, in `WebApplication.Extensions.cs`, import the namespace for security claims, as shown in the following code:

```
using System.Security.Claims; // To use
ClaimsPrincipal.
```

6. In `WebApplication.Extensions.cs`, after mapping an HTTP GET request for the root path to return a plain text `Hello World` response, add a statement to map an HTTP GET request for the secret path to return the authenticated user's name if they are authorized, as shown in the following code:

```
app.MapGet("/", () => "Hello World!")
    .ExcludeFromDescription(); app.MapGet("/
secret", (ClaimsPrincipal user) =>
string.Format("Welcome, {0}. The secret
ingredient is love.", user.Identity?.Name
?? "secure user"))
```



```
.RequireAuthorization();
```

- ```
dotnet user-jwts create
```

- ```
New JWT saved with ID 'd7e22000'. Name:  
markjprice Token:  
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1bmVmlx
```

- ```
dotnet user-jwts print d7e22000 --show-all
```

- ```
Found JWT with ID 'd7e22000'. ID: d7e22000
Name: markjprice Scheme: Bearer
Audience(s): http://localhost:5100,
```

```
https://localhost:5101 Not Before:
2025-09-26T10:58:18.0000000+00:00 Expires
On: 2025-12-26T10:58:18.0000000+00:00
Issued On:
2025-09-26T10:58:19.0000000+00:00 Scopes:
none Roles: [none] Custom Claims: [none]
Token Header: {"alg":"HS256","typ":"JWT"}
Token Payload:
{"unique_name":"markjprice","sub":"markjprice",
["http://localhost:5100","https://
localhost:5101"],"nbf":1664189898,"exp":167205229
user-jwts"} Compact Token:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1bm90aWQiOiJm
arkjprice","sub":"markjprice",
```

11. In the `Northwind.WebApi.Service` project, in `appsettings.Development.json`, note the new section named `Authentication`, as shown highlighted in the following configuration:

```
{ "Logging": { "LogLevel": { "Default":  
"Information", "Microsoft.AspNetCore":  
"Warning",  
"Microsoft.AspNetCore.HttpLogging":  
"Information" } }, ... "Authentication" : {  
"Schemes" : { "Bearer" : { "ValidAudiences"  
: [ "http://localhost:5100" "https://  
localhost:5101" ], "ValidIssuer" "dotnet-  
user-jwts": } } } }
```

12. Start the `Northwind.WebApi.Service` project using the `https` profile without debugging.
13. In the browser, change the relative path to `/secret` and note the response is rejected with a 401 status code.

14. Start VS Code and open the `HttpRequests` folder.
15. In the `HttpRequests` folder, create a file named `webapi-secure-request.http` and modify its contents to contain a request to get the secret ingredient, as shown in the following code (but use your Bearer token, of course):

```
### Get the secret ingredient. GET
https://localhost:5101/secret/
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1bm1xdWVf
```

16. Click **Send Request**, and note the response, as shown in the following output:

```
Welcome, secure user. The secret
ingredient is love.
```

17. Close the browser and shut down the web service.

Practicing and exploring

Test your knowledge and understanding by answering some questions, getting some hands-on practice, and exploring this chapter's topics with deeper research.

Exercise 10.1 – Online material

Online material can be extra content written by me for this book, or it can be references to content created by Microsoft or third parties.

What's New in ASP.NET Core 10 Minimal API

A summary of what's new in .NET 10 for Minimal API web services can be found at the following link: <https://github.com/markjprice/markjprice/blob/main/articles/whats-new-in-net10-books.md#chapter-15---building-and-consuming-web-services>

Support for Server-Sent Events (SSE)

ASP.NET Core can now return Server-Sent Events using the `TypedResults.ServerSentEvents` API. This capability works in both Minimal API and Web API with controllers.

Server-Sent Events, or SSE, is a server push mechanism that lets a server continuously send event messages to a client over a single HTTP connection. In .NET, each event is represented by an `SseItem<T>` instance, which can include an event name, an event ID, and a strongly typed data payload.

The `TypedResults` class introduces a new static `ServerSentEvents` method for producing an SSE response. This method accepts an `IAsyncEnumerable<SseItem<T>>` as its first argument, which represents the asynchronous stream of events that will be delivered to the client.

You can learn more about SSE support in ASP.NET Core 10 at the following link:

<https://learn.microsoft.com/en-gb/aspnet/core/release-notes/aspnetcore-10.0?view=aspnetcore-10.0#support-for-server-sent-events-sse>

Review Microsoft HTTP API design policy

Microsoft has internal HTTP/REST API design guidelines. Microsoft teams reference this document when designing their HTTP APIs. They are a great starting point for your own standards for HTTP APIs. You can review them at the following link: <https://github.com/microsoft/api->

[guidelines](#).

The guidelines have a section on CORS, which you can review at the following link: <https://github.com/microsoft/api-guidelines/blob/vNext/graph/Guidelines-deprecated.md#8-cors>.

Exercise 10.2 – Practice exercises

Visual Studio scaffolding for Minimal API web services

It is important to learn how to implement a service using Minimal API from scratch so that you properly understand it. But once you know how to do it manually, the process can be automated and the boilerplate code can be written for you, especially if you are building a web API that wraps an EF Core entity model.

For example, Visual Studio has a project item template named **API with read/write endpoints, using Entity Framework** that allows you to select:

- An entity model class like `Customer`.
- An endpoints class that will contain all the mapping methods.
- A `DbContext`-derived class like `NorthwindContext`.
- A database provider like SQLite or SQL Server.

This template saves time, removes boring setup work, and quietly nudges developers toward patterns that are hard to mess up later. It produces a controller with REST-style endpoints for `GET` all, `GET` by id, `POST`, `PUT`, and `DELETE` with `async` methods everywhere and proper HTTP status codes like 404, 201, and 204, and model binding and validation attributes already in place.

You can learn more about this project item template at the following link: <https://>

```
devblogs.microsoft.com/  
visualstudio/web-api-development-  
in-visual-studio-2022/  
#scaffolding-in-visual-studio/.
```

Auth0 project templates

If you need to implement Auth0 for authentication and authorization, then you can use project templates to scaffold your code. An article describing these project templates and how to use them is found at the following link:

<https://auth0.com/blog/introducing-auth0-templates-for-dotnet/>.

Exercise 10.3 – Test your knowledge

Answer the following questions:

1. List six method names that can be specified in an HTTP request.
2. List six status codes and their descriptions that can be returned in an HTTP response.
3. How is the ASP.NET Core Minimal API web service technology different from the ASP.NET Core Web API service technology?
4. With the ASP.NET Core Minimal API web service technology, how do you map an HTTP PUT request to `api/customers` to a lambda statement block?
5. With the ASP.NET Core Minimal API web service technology, how do you map a method or lambda parameter to a value in a route, query string, or the body of the request?
6. Does enabling CORS increase security for a web service?
7. You have added statements to `Program.cs` to enable HTTP logging but HTTP requests and responses are not being logged. What is the most likely reason and how can you fix it?
8. How do you limit the rate of requests for a specific client using the

AspNetCoreRateLimit package?

9. How do you limit the rate of requests for a specific endpoint using the `Microsoft.AspNetCore.RateLimiting` package?

10. What does JWT mean?

Exercise 10.4 – Explore topics

Use the links on the following page to learn more details about the topics covered in this chapter: <https://github.com/markjprice/apps-services-net10/blob/main/docs/book-links.md#chapter-10---building-and-securing-minimal-api-web-services>.

Summary

In this chapter, you learned how to:

- Build a web service that implements the REST architectural style using ASP.NET Core Minimal API.
- Relax the same-origin security policy for specified domains and ports using CORS.
- Implement two different rate-limiting packages to prevent denial of service attacks.
- Secure services using JWT bearer authorization.

In the next chapter, you will learn how to build reliable and scalable services by adding features like caching, queues, and automatic handling of transient faults using libraries like Polly.

Get This Book **UNLOCK NOW** and Exclusive Extras

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.



Note: *Keep your invoice handy. Purchases made directly from Packt don't require one.*

11

Caching, Queuing, and Resilient Background Services

In this chapter, you will be introduced to multiple technologies and techniques that will improve the scalability and reliability of your services, no matter what service technology you choose to implement them with.

This chapter will cover the following topics:

- Understanding service architecture
- Caching with ASP.NET Core
- Fault tolerance with Polly
- Queuing with RabbitMQ
- Implementing long-running background services

Understanding service architecture

In [Chapter 10](#), *Building and Securing Minimal API Web Services*, you learned how to build a web service using ASP.NET Core Minimal API. Before looking at alternative technologies to build services, it is worth taking a step back and reviewing service architecture and what causes bottlenecks in service performance and scalability.

What parts of a system are the slowest?

Traditionally, the slowest parts of a system are:

- The network (slowest)
- The disk
- Memory
- CPU cache memory (fastest)

Each layer can be 5 to 10 times slower than the next.

However, networks are much faster than they used to be, and systems often run within remote data centers. Imagine your service needs some data. Is it faster to read from the local server disk or to make a call to another server?

- Server-to-server call within the same data center: 500,000 nanoseconds (ns)
- Disk seek: 10,000,000 ns

Numbers every developer should know

Jeff Dean, a Google Fellow, quotes in his presentations the actual times in nanoseconds (ns) for various technologies to access or read data. They are shown in *Table 11.1*, and I have added a column to scale the numbers to be more comprehensible to a human:

Technology	Time (ns)	Human Scale
Cache hit	10	0.00000001s
Cache miss	100	0.0000001s
Local memory reference	100	0.0000001s
Local disk reference	1,000,000	0.001s
Network lock/unlock	1,000,000	0.001s
Remote memory reference	1,000,000	0.001s
Remote disk reference	10,000,000	0.01s
Remote byte over 1 Gbps network	10,000,000	0.01s
Remote disk sequentially from memory	10,000,000	0.01s
Remote disk within same datacenter	10,000,000	0.01s

Read 100MB sequentially from SSD					
Read 1GB sequentially from SSD					
Read 100MB sequentially from disk					
Read 1GB sequentially from disk					
Stream 1GB/s CA->Netherlands->CA					

Table 11.1: Nanosecond times for various technologies to access or read data

Designs, Lessons, and Advice from Building Large Distributed Systems, by Jeff Dean, <http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf>.

The point is not to debate if reading from drives or SSDs is faster or not than network calls. It is more about being aware of the differences between getting data that’s close and data that’s far away. The orders of magnitude are comparatively massive.

For example, if a CPU needs to process some data and it is already in the L1 cache, then it takes the equivalent of 1 second. If it needs to read the data from memory, it takes the equivalent of a quarter of an hour. If it needs to fetch that data from a server in the same data center, it takes the equivalent of 2 months. And if it is in California and the data is in the Netherlands, it takes the equivalent of 47½ years!

This is why caching is so important. Caching is about temporarily storing data as close to where it will be needed as possible.

Caching with ASP.NET Core

Caching can enable our systems to copy some data from a remote data center to a local data center, or from a server or disk to memory. Caches store data as key-value pairs.

However, one of the hardest parts of caching is getting the balance right

between storing enough data and keeping it fresh. The more data we copy, the more resources we use. We also need to consider how we will keep the copies synchronized with the original data.

General caching guidelines

Caching works best with data that costs a lot to generate and does not change often.

Follow these guidelines when caching:

- Your code should never depend on cached data. It should always be able to get the data from the original source when the data is not found in the cache.
- Wherever you cache data (in-memory or in a database) it is a limited resource, so deliberately limit the amount of data cached and for how long by implementing expirations and size limits. You should monitor cache hits (when data is successfully found in the cache) to obtain the right balance for your specific scenarios.

In the coding tasks in this section, you will implement all of these guidelines.

Let's start by reviewing the caching technologies built into ASP.NET Core.

Building a controller-based Web API service

To explore various caching technologies, let's build a basic web service:

1. Use your preferred code editor to create a new Web API controller-based project, as defined in the following list:
 - Project template: **ASP.NET Core Web API** / `webapi --use-controllers`
 - Solution file and folder: `ModernServices`
 - Project file and folder:

Northwind.WebApi.Controllers

- **Authentication type:** None
- **Configure for HTTPS:** Selected
- **Enable container support:** Cleared
- **Use controllers:** Selected
- **Enable OpenAPI support:** Selected
- **Do not use top-level statements:** Cleared
- **Enlist in Aspire orchestration:** Cleared

Make sure to select the **Use controllers** checkbox or specify the `--use-controllers` or `-controllers` switch. We will not use minimal APIs, which is the default way a Web API is implemented using the .NET 8 or later project templates. If you use Rider, you might want to use the `dotnet new` command until Rider supports a **Use controllers** option.

1. Add a project reference to the Northwind database context project, as shown in the following markup:

```
<ItemGroup> <ProjectReference Include= "..  
  \..\ModernApps\Northwind.DataContext  
  \Northwind.DataContext.csproj" /> </  
ItemGroup>
```

The path cannot have a line break. If you did not complete the task of creating the class libraries in [Chapter 1, Introducing Apps and Services with .NET](#), then download the

solution projects from the GitHub repository.

1. In the project file, treat warnings as errors, as shown in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup> <TargetFramework>net10.0</
  TargetFramework> <Nullable>enable</
  Nullable> <ImplicitUsings>enable</
  ImplicitUsings>
  <TreatWarningsAsErrors>true</
  TreatWarningsAsErrors> </PropertyGroup>
```

2. Build the Northwind.WebApi.Controllers project to make sure the entity model class library projects are properly compiled and their DLLs copied to the local project's bin folder.
3. In the Properties folder, in launchSettings.json, modify the applicationUrl of the profile named https to use port 5111 for https and port 5112 for http, as highlighted in the following configuration:

```
"profiles": { ... "https" :{
  "commandName": "Project",
  "dotnetRunMessages": true,
  "launchBrowser": true, "applicationUrl"
  "https://localhost:5111;http://
  localhost:5112":, "environmentVariables":
  { "ASPNETCORE_ENVIRONMENT": "Development"
  }
}
```

Visual Studio and Rider will read this

settings file and automatically run a web browser if `launchBrowser` is `true`, and then navigate to the `applicationUrl` and `launchUrl`. VS Code and `dotnet run` will not, so you will need to run a web browser and navigate manually to <https://localhost:5111/>.

1. Delete the file named `WeatherForecast.cs`.
2. In the `Controllers` folder, delete the file named `WeatherForecastController.cs`.
3. In `Program.cs`, import the namespace to add the `NorthwindContext` to the configured services, as highlighted in the following code:

```
using Northwind.EntityModels; // To use  
the AddNorthwindContext method. var  
builder =  
WebApplication.CreateBuilder(args); // Add  
services to the container.  
builder.Services.AddNorthwindContext();  
builder.Services.AddControllers();
```

4. In the `Controllers` folder, add a new class file named `ProductsController.cs`.
5. In `ProductsController.cs`, modify its contents to define a controller-based Web API to work with products in the `Northwind` database, as we did for minimal APIs, as shown in the following code:

```
using Microsoft.AspNetCore.Mvc; // To use  
[HttpGet] and so on. using
```

```

Northwind.EntityModels; // To use
NorthwindContext, Product. namespace
Northwind.WebApi.Service.Controllers;
[Route("api/products")] [ApiController]
public class ProductsController :
ControllerBase { private int pageSize =
10; private readonly NorthwindContext _db;
public ProductsController(NorthwindContext
context) { _db = context; } // GET: api/
products [HttpGet]
[Produces(typeof(Product[]))] public
IEnumerable<Product> Get(int? page) {
return _db.Products .Where(p =>
p.UnitsInStock > 0 && !p.Discontinued)
.OrderBy(product => product.ProductId)
.Skip((page ?? 1) - 1) * pageSize)
.Take(pageSize); } // GET: api/products/
outofstock [HttpGet] [Route("outofstock")]
[Produces(typeof(Product[]))] public
IEnumerable<Product>
GetOutOfStockProducts() { return
_db.Products .Where(p => p.UnitsInStock ==
0 && !p.Discontinued); } // GET: api/
products/discontinued [HttpGet]
[Route("discontinued")]
[Produces(typeof(Product[]))] public
IEnumerable<Product>
GetDiscontinuedProducts() { return
_db.Products .Where(product =>
product.Discontinued); } // GET api/
products/5 [HttpGet("{id:int}")] public
async ValueTask<Product?> Get(int id) {
return await _db.Products.FindAsync(id); }
// GET api/products/cha
[HttpGet("{name}")] public

```



```

IEnumerable<Product> Get(string name) {
    return _db.Products.Where(p =>
        p.ProductName.Contains(name)); } // POST
api/products [HttpPost] public async
Task<IActionResult> Post([FromBody]
Product product) {
    _db.Products.Add(product); await
    _db.SaveChangesAsync(); return
    Created($"api/products/
    {product.ProductId}", product); } // PUT
api/products/5 [HttpPut("{id}")] public
async Task<IActionResult> Put(int id,
    [FromBody] Product product) { Product?
    foundProduct = await
    _db.Products.FindAsync(id); if
    (foundProduct is null) return NotFound();
    foundProduct.ProductName =
    product.ProductName;
    foundProduct.CategoryId =
    product.CategoryId;
    foundProduct.SupplierId =
    product.SupplierId;
    foundProduct.QuantityPerUnit =
    product.QuantityPerUnit;
    foundProduct.UnitsInStock =
    product.UnitsInStock;
    foundProduct.UnitsOnOrder =
    product.UnitsOnOrder;
    foundProduct.ReorderLevel =
    product.ReorderLevel;
    foundProduct.UnitPrice =
    product.UnitPrice;
    foundProduct.Discontinued =
    product.Discontinued; await
    _db.SaveChangesAsync(); return

```

```

NoContent(); } // DELETE api/products/5
[HttpDelete("{id}")] public async
Task<IActionResult> Delete(int id) { if
(await _db.Products.FindAsync(id) is
Product product) {
_db.Products.Remove(product); await
_db.SaveChangesAsync(); return
NoContent(); } return NotFound(); } }

```

6. If your database server is not running, for example, because you are hosting it in Docker, a virtual machine, or the cloud, then make sure to start it.
7. Start the web service project using the `https` profile without debugging.
 - If you are using Visual Studio, select the **https** profile in the drop-down list, and then navigate to **Debug | Start Without Debugging** or press `Ctrl + F5`.
 - If you are using VS Code, enter the command `dotnet run --launch-profile https`, manually start a web browser.
10. Navigate to the OpenAPI JSON document: <https://localhost:5111/openapi/v1.json>.

On Windows, if prompted to do so, you will have to set the Windows Defender Firewall to allow access to your local web service.

1. Use the `.http` files we created to test the Minimal API web service in [Chapter 10, Building and Securing Minimal API Web Services](#) to test the various endpoints, but change the port from `5101` to `5111`, and note the SQL statements logged to the output as you do so, for example:
 - Get the first page of 10 products.

- Get the 6th page of 10 products.
- Get the product with an ID of 77.
- Get the single out-of-stock product.
- Get the seven discontinued products.
- Get products whose names start with `cha`.
- Create (POST), update (PUT), and delete a product.

Now that we have a basic web service, we can start activating caching in it.

Caching objects using in-memory caching

The `IMemoryCache` interface represents a cache that uses local server memory. If you have multiple servers hosting your service or website, then you must enable “sticky sessions,” although this is an anti-pattern with Kubernetes. This means that an incoming request from a client or visitor will be directed to the same server as previous requests from that client or visitor, allowing the request to find the correct cached data in that server’s memory.

The `Microsoft.Extensions.Caching.Memory` package has a modern implementation of `IMemoryCache`. Avoid the older `System.Runtime.Caching`.

Sizes are defined using custom units. If you store simple `string` values, then you could use the length of the `string` value. If you don’t know the size, you could just use 1 unit for each entry to simply limit the number of entries.

When you add an object to a cache, you should set an expiration. There are two types, absolute and sliding, and you can set one or the other, both, or neither:

- **Absolute expiration:** This is a fixed date/time, for example, 1am on December 24, 2023. When the date/time is reached, the object is evicted. To use this, set the `AbsoluteExpiration` property of a cache entry to a `DateTime` value. Choose this if you need to guarantee that at some point the data in the cache will be refreshed.

- **Sliding expiration:** This is a time span, for example, 20 seconds. When the time span expires, the object is evicted. However, whenever an object is read from the cache, its expiration is reset for another 20 seconds. This is why it is described as *sliding*. A common duration for a **Content Management System (CMS)**, where content like a web page is loaded from a database, is 12 hours. Content frequently viewed by visitors, like the home page, is then likely to remain in memory. To use this, set the `SlidingExpiration` property of a cache entry to a `TimeSpan` value. Choose this if it is acceptable for data to potentially never be refreshed. A good CMS will have an additional mechanism to reliably force a refresh when new content is published, but this functionality is not built into .NET caching.
- **Both expirations:** If you only set a sliding expiration, an object may stay in the cache forever, so you might also want to set the `AbsoluteExpirationRelativeToNow` property to a `TimeSpan` further in the future, after which the object should definitely be evicted. Choose this if you want the best of both worlds.
- **Never:** You can set a cache entry to have a priority of `CacheItemPriority.NeverRemove`.

You can also configure a method to call back to when an object is evicted from the cache. This allows you to execute some business logic to decide if you want to add the object back into the cache, perhaps after refreshing it from the original data source. You do this by calling the `RegisterPostEvictionCallback` method.

Let's explore the in-memory cache:

1. In the `Northwind.WebApi.Controllers` project, in `Program.cs`, import the namespace to work with the in-memory cache, as shown in the following code:

```
using Microsoft.Extensions.Caching.Memory;  
// To use IMemoryCache and so on.
```

2. In `Program.cs`, after the call to `CreateBuilder`, in the section for configuring services, register an implementation for the in-memory cache, configured to store a maximum of 50 products, as shown in the following code:

```
builder.Services.AddSingleton<IMemoryCache>(new
MemoryCache( new MemoryCacheOptions {
TrackStatistics = true, SizeLimit = 50 //
Products. }));
```

3. In `ProductsController.cs`, import the namespace to work with the in-memory cache, as shown in the following code:

```
using Microsoft.Extensions.Caching.Memory;
// To use IMemoryCache.
```

4. In `ProductsController.cs`, declare some fields to store a logger, the in-memory cache, and a key for the out-of-stock products, and set them in the constructor, as shown in the following code:

```
private readonly
ILogger<ProductsController> logger;
private readonly IMemoryCache
_memoryCache; private const string
OutOfStockProductsKey = "OOSP"; public
ProductsController( NorthwindContext
context, ILogger<ProductsController>
logger, IMemoryCache memoryCache) { _db =
context; _logger = logger; _memoryCache =
memoryCache; }
```

5. In `ProductsController.cs`, in the `GetOutOfStockProducts` action method, add statements to try to get the cached out-of-stock products, and if they are not cached, get them from the database and set them in the cache, using a sliding expiration of five seconds, as highlighted in the following code:

```
// GET: api/products/outofstock [HttpGet]
[Route("outofstock")]
[Produces(typeof(Product[]))] public
IEnumerable<Product>
GetOutOfStockProducts() { // Try to get
the cached value. if (!
_memoryCache.TryGetValue(OutOfStockProductsKey,
out Product[]? cachedValue)) { // If the
cached value is not found, get the value
from the database. cachedValue =
_db.Products .Where(p => p.UnitsInStock ==
0 && !p.Discontinued) .ToArray();
MemoryCacheEntryOptions cacheEntryOptions
= new() { SlidingExpiration =
TimeSpan.FromSeconds(5), Size =
cachedValue?.Length };
_memoryCache.Set(OutOfStockProductsKey,
cachedValue, cacheEntryOptions); }
MemoryCacheStatistics? stats =
_memoryCache.GetCurrentStatistics();
_logger.LogInformation("Memory cache.
Total hits: {stats?.TotalHits}. Estimated
size: { stats?.CurrentEstimatedSize}.");
return cachedValue ??
Enumerable.Empty<Product>(); }
```

6. Start the web service project using the `https` profile without debugging.

7. Arrange the windows so that you can see the command prompt or terminal at the same time as the web page.
8. Navigate to `/api/product/outofstock` and note in the output that EF Core executes a SQL statement to get the products, the total hit counter is zero, and one product has now been cached, as shown in the following output:

```
info:  
Northwind.WebApi.Service.Controllers.ProductsCont  
Memory cache. Total hits: 0. Estimated  
size: 1.
```

9. In your browser, click **Refresh** within five seconds, and continue to click it a few more times:
 - Note that EF Core does not need to re-execute the SQL statement because the products are cached, and if something reads them within a five-second sliding expiration, they will stay in memory forever.
 - Note the total hit counter for the cache increments each time the out-of-stock products are found in the cache, as shown in the following output:

```
info:  
Northwind.WebApi.Service.Controllers.ProductsCont  
Memory cache. Total hits: 1. Estimated  
size: 1. info:  
Northwind.WebApi.Service.Controllers.ProductsCont  
Memory cache. Total hits: 2. Estimated  
size: 1. info:  
Northwind.WebApi.Service.Controllers.ProductsCont  
Memory cache. Total hits: 3. Estimated  
size: 1.
```

12. Wait at least five seconds.
13. Click **Refresh**, and note in the output that EF Core executes a SQL statement to get the products because they have not been read within the five-second sliding expiration window.
14. Close the browser and shut down the web server.

Caching objects using distributed caching

Distributed caches have benefits over in-memory caches. Cached objects:

- Are consistent across requests to multiple servers.
- Survive server restarts and service deployments.
- Do not waste local server memory.
- Are stored in a shared area, so in a server farm scenario with multiple servers, you do not need to enable sticky sessions.

Warning! A disadvantage of distributed caches is that in-memory caches can store any object, but a distributed cache can only store byte arrays. Your object needs to be serialized and sent across the network to the remote cache.

Microsoft provides the `IDistributedCache` interface with pre-defined methods to manipulate items in any distributed cache implementation. The methods are:

- `Set` or `SetAsync`: To store an object in the cache.
- `Get` or `GetAsync`: To retrieve an object from the cache.
- `Remove` or `RemoveAsync`: To remove an object from the cache.
- `Refresh` or `RefreshAsync`: To reset the sliding expiration for an object in the cache.

There are many implementations of distributed caching to choose from, including the following:

- SQL Server: <https://learn.microsoft.com/en-us/aspnet/core/performance/caching/distributed#distributed-sql-server-cache>
- Redis: <https://learn.microsoft.com/en-us/aspnet/core/performance/caching/distributed#distributed-redis-cache>
- NCache: <http://www.alachisoft.com/ncache/aspnet-core-idistributedcache-ncache.html>

We will use the **Distributed Memory Cache**, which is a Microsoft built-in implementation of `IDistributedCache` that stores items in memory on the server where the service runs.

It is not an actual distributed cache, but it is useful for scenarios like unit testing, where you want to remove the dependency on yet another external service, or while learning, as we are doing in this book.

Later, you only need to change the configured distributed cache, not the service implementation code that uses it, because all interactions go through the registered `IDistributedCache` implementation.

Let's go!

1. In the `Northwind.WebApi.Controllers` project, in `Program.cs`, after the call to `CreateBuilder`, in the section for configuring services, register the implementation for the distributed memory cache, as shown in the following code:

```
builder.Services.AddDistributedMemoryCache();
```

2. In `ProductsController.cs`, import the namespace for working with a distributed cache implementation and serialized JSON, as shown

in the following code:

```
using
Microsoft.Extensions.Caching.Distributed;
// To use IDistributedCache. using
System.Text.Json; // To use
JsonSerializer.
```

3. In `ProductsController.cs`, declare some fields to store the distributed cache implementation and an item key for discontinued products, as highlighted in the following code:

```
private readonly IMemoryCache
_memoryCache; private const string
OutOfStockProductsKey = "OOSP"; private
readonly IDistributedCache
_distributedCache; private const string
DiscontinuedProductsKey = "DISCP"; public
ProductsController(
ILogger<ProductsController> logger,
NorthwindContext context, IMemoryCache
memoryCache, IDistributedCache
distributedCache) { _logger = logger; _db
= context; _memoryCache = memoryCache;
_distributedCache = distributedCache; }
```

4. In `ProductsController.cs`, define a private method to get the discontinued products from the database, and set them in the distributed cache, using a sliding expiration of 5 seconds and an absolute expiration of 20 seconds, as shown in the following code:

```
private Product[]?
```

```

GetDiscontinuedProductsFromDatabase() {
    Product[]? cachedValue = _db.Products
        .Where(product => product.Discontinued)
        .ToArray(); DistributedCacheEntryOptions
        cacheEntryOptions = new() { // Allow
            readers to reset the cache entry's
            lifetime. SlidingExpiration =
            TimeSpan.FromSeconds(5), // Set an
            absolute expiration time for the cache
            entry. AbsoluteExpirationRelativeToNow =
            TimeSpan.FromSeconds(20), }; byte[]?
            cachedValueBytes =
            JsonSerializer.SerializeToUtf8Bytes(cachedValue);
            _distributedCache.Set(DiscontinuedProductsKey,
            cachedValueBytes, cacheEntryOptions);
            return cachedValue; }

```

5. In ProductsController.cs, in the

GetDiscontinuedProducts action method, add statements to try to get the cached discontinued products, and if not cached, get them from the database. If a byte array is found in the cache, try to deserialize it into products, but if that fails too, get the products from the database, as highlighted in the following code:

```

// GET: api/products/discontinued
[HttpGet] [Route("discontinued")]
[Produces(typeof(Product[]))] public
IEnumerable<Product>
GetDiscontinuedProducts() { // Try to get
    the cached value. byte[]? cachedValueBytes
    =
    _distributedCache.Get(DiscontinuedProductsKey);
    Product[]? cachedValue = null; if
    (cachedValueBytes is null) { cachedValue =

```

```
GetDiscontinuedProductsFromDatabase(); }  
else { cachedValue = JsonSerializer  
.Deserialize<Product[]?>  
>(cachedValueBytes); if (cachedValue is  
null) { cachedValue =  
GetDiscontinuedProductsFromDatabase(); } }  
return cachedValue ??  
Enumerable.Empty<Product>(); }
```

Unlike the in-memory cache that can store any live object, objects stored in distributed cache implementations must be serialized into byte arrays because they need to be transmittable across networks.

1. Start the web service project, using the `https` profile without debugging.
2. Arrange the windows so that you can see the command prompt or terminal at the same time as the web page.
3. Navigate to `/api/product/discontinued`, and note in the output that EF Core executes a SQL statement to get the products.
4. Click **Refresh** within five seconds, continue to click it a few more times, and note that EF Core does not need to re-execute the SQL statement because the products are cached. If something reads them within a five-second sliding expiration, they will stay in memory forever.
5. Wait at least five seconds.
6. Click **Refresh**, and note in the output that EF Core executes a SQL statement to get the products because they have not been read within the five-second sliding expiration window.
7. Continue to click **Refresh** repeatedly, and note that after 20 seconds, EF Core must execute a SQL statement to refresh the products.
8. Close the browser and shut down the web server.

You have now implemented both in-memory and distributed object caching. Wouldn't it be nice not to have to choose? Wouldn't it be nice to have object caching that is smart enough to use the best of both automatically? Well, now you can.

Hybrid object caching

The `HybridCache` API, introduced in preview with ASP.NET Core 9 and now fully available with ASP.NET Core 10, addresses some limitations found in the `IDistributedCache` and `IMemoryCache` APIs. As an abstract class with a default implementation, `HybridCache` efficiently manages most tasks related to storing and retrieving data from the cache.

The key points about hybrid caching are shown in the following list:

- **Unified API:** It provides a single interface for both in-memory (aka in-process) and distributed (aka out-of-process) object caching. `HybridCache` can seamlessly replace any existing `IDistributedCache` and `IMemoryCache` usage. It always uses the in-memory cache initially, and when an `IDistributedCache` implementation is available, `HybridCache` leverages it for secondary caching. This dual-level caching approach combines the speed of in-memory caching with the durability of distributed or persistent caching.
- **Stampede protection:** `HybridCache` prevents cache stampedes, which occur when a frequently used cache entry is invalidated, causing multiple requests to try to repopulate it simultaneously. `HybridCache` merges concurrent operations, ensuring all requests for the same response wait for the first request to complete.
- **Configurable serialization:** `HybridCache` allows for configurable serialization during service registration, supporting both type-specific and generalized serializers via the `WithSerializer` and `WithSerializerFactory` methods, which are chained from the `AddHybridCache` call. By default, it manages `string` and

`byte[]` internally and utilizes `System.Text.Json` for other types. It can be configured to use other serializers, such as Protobuf or XML.

Although `HybridCache` was introduced in preview with .NET 9, its package targets .NET Standard 2.0, so it can be used with older versions of .NET, even including .NET Framework 4.6.2 or later. The `HybridCache` package reached general availability in March 2025 with version 9.3 (previous versions were all previews): <https://devblogs.microsoft.com/dotnet/hybrid-cache-is-now-ga/>.

Enabling hybrid caching in an ASP.NET Core project

To enable hybrid caching in an ASP.NET Core project:

1. Add a package reference for hybrid caching, as shown in the following markup:

```
<PackageReference
  Include="Microsoft.Extensions.Caching.Hybrid"
/>
```

2. In `Program.cs`, import the namespace for working with a hybrid cache, as shown in the following code:

```
using Microsoft.Extensions.Caching.Hybrid;
// To use HybridCacheEntryOptions.
```

3. In `Program.cs`, after the call to `CreateBuilder`, in the section for

configuring services, register the hybrid cache service with appropriate options like a default cache entry duration of 60 seconds overall and 30 seconds for local in-memory caching, as shown in the following code:

```
builder.Services.AddHybridCache(options =>
{ options.DefaultEntryOptions = new
HybridCacheEntryOptions { Expiration =
TimeSpan.FromSeconds(DurationInSeconds.OneMinute)
LocalCacheExpiration =
TimeSpan.FromSeconds(DurationInSeconds.OneMinute)
}; });
```

4. In a class that needs caching, get and store the hybrid cache, as shown in the following code:

```
using Microsoft.Extensions.Caching.Hybrid;
// To use HybridCache. ... private
readonly HybridCache _cache; ... public
CustomerRepository(HybridCache
hybridCache, ...) { _cache = hybridCache;
...}
```

5. Get an object like a customer from the cache if possible or from the data model and set it in the cache for next time, as shown in the following code:

```
Customer customer = await
_cache.GetOrCreateAsync( key: id, //
Unique key to the cache entry. factory:
async cancel => await _db.Customers
.FirstOrDefaultAsync(c => c.CustomerId ==
id, token), cancellationToken: token);
```

What's in my object cache?

I sometimes get asked by readers how to see what is currently stored in an object cache. Cache APIs like `IMemoryCache` in .NET typically do not provide a built-in method to view all items in the cache due to a combination of performance considerations, design philosophy, and intended use cases. Caches are optimized for high-speed lookups rather than iteration. Allowing access to all cached items would require maintaining an enumerable collection of keys or values.

Caches often work in multithreaded environments. Providing a way to enumerate all cached items introduces synchronization challenges. Applications are expected to request specific items by key, not to query or explore the cache as a data store.

If enumeration is required, it should be implemented explicitly by the application, not as a built-in feature of the cache. This forces developers to understand the trade-offs of maintaining their own key-tracking mechanism, and design custom enumeration solutions that align with their application's requirements, such as creating a wrapper around `IMemoryCache` that stores keys separately.

If your application requires visibility into cached items, it's a sign you might need to rethink how you're using the cache. Caches are not intended to replace databases or general-purpose collections.

Caching web responses using HTTP caching

In-memory and distributed caching work with any type of app or service, using any transport technology, because all the magic happens on the server.

HTTP caching aka response caching is tied to HTTP GET requests and responses because it is based on HTTP headers. Therefore, it only works with

apps and services that use HTTP as their transport technology, like web services built using ASP.NET Core Web API with controllers, Minimal API, and OData.

You can read the official standard for HTTP caching at the following link: <https://www.rfc-editor.org/rfc/rfc9111>.

Requirements for HTTP caching aka response caching include the following:

- The request must be `GET` or `HEAD`. `POST`, `PUT`, and `DELETE` requests, and so on, are never cached by HTTP caching.
- The response must have a `200 OK` status code.
- If the request has an `Authorization` header, then the response is not cached.
- If the request has a `Vary` header, then the response is not cached when the values are not valid or `*`. Valid values include `User-Agent`, `Accept-Encoding`, `Accept-Language`, and `Cookie`.

The web server sets response caching headers, and then intermediate proxies and clients should read those headers to see how they should cache the responses.

Good practice: HTTP caching aka response caching is not typically useful for web user interfaces because web browsers often set request headers that prevent HTTP caching. For web user interfaces, output caching is better suited.

The `Cache-Control` HTTP header for requests and responses has some common directives, as shown in *Table 11.2*:

Discription		
6 clients and intermediaries can cache this response		
Only a client should cache this response		

The client does not accept responses older than the specified number of seconds		
A client request is asking for a non-cached response. A server is telling the client and intermediaries not to cache the response		
A cache must not store the request or response.		

Table 11.2: Common Cache-Control HTTP header directives

As well as Cache-Control, there are other headers that might affect caching, as shown in Table 11.3:

Header	Description
Expires	Estimated number of seconds since the response was generated
Cache-Control: no-cache	Absolute date/time after which the response should be considered expired
Cache-Control: no-store	All fields must match for a cached response to be sent. Otherwise, a fresh response is sent. For example, a query string of color.

Table 11.3: Common HTTP headers for caching

For example, a client could ask for a fresh list of discontinued products, and the service should not use any cached version, as shown in the following HTTP response:

```
GET api/products/discontinued Cache-Control:
no-cache
```

A service could return some products as a JSON array, with a header to say that intermediaries should not cache the response, but clients can, as shown in the following HTTP response:

```
content-type: application/json; charset=utf-8
date: Fri, 09 Jun 2023 06:05:13 GMT server:
Kestrel cache-control: private [ {
```

```
"productId": 5, "productName": "Chef Anton's  
Gumbo Mix", ...
```

Decorate a controller or method with the `[ResponseCache]` attribute to control caching responses from the server (code to control caching requests has to go in the client code). This attribute has common parameters, as shown in *Table 11.4*:

Description		
How long to cache in seconds		
When the response can be cached.	Any (cache-control: public), Client (cache-control: private), None (cache-control: no-cache)	
Set cache-control: no-store		
Set the Vary header		
QueryKeys to vary by.		

Table 11.4: Common parameters of the `[ResponseCache]` attribute

Let's apply response caching to the web service:

1. In the `Northwind.WebApi.Controllers` project, in `Program.cs`, after the call to add the distributed memory cache, add a statement to add response caching middleware as a dependency service, as shown in the following code:

```
builder.Services.AddResponseCaching();
```

2. In `Program.cs`, after the call to use HTTPS redirection, add a statement to use response caching middleware, as shown in the following code:

```
app.UseResponseCaching();
```

Good practice: If using CORS middleware, then `UseCors` must be called before `UseResponseCaching`.

1. In `ProductsController.cs`, decorate the `Get` method with an `int id` parameter with the `[ResponseCache]` attribute, as highlighted in the following code:

```
// GET api/products/5
[HttpGet("{id:int}")]
[ResponseCache(Duration = 5, // Cache-
Control: max-age=5 Location =
ResponseCacheLocation.Any, // Cache-
Control: public VaryByHeader = "User-
Agent" // Vary: User-Agent )] public async
ValueTask<Product?> Get(int id) { return
await _db.Products.FindAsync(id); }
```

The `[ResponseCache]` attribute can be applied to Razor Pages, MVC controller classes, and MVC action methods for both web services and websites.

1. Start the web service project, using the `https` profile without debugging.
2. In the `HttpRequests` folder, open the `webapi-get-products.http` file.
3. Modify the base address to use port `5111`, and then send the request for a specific product, like `77`, as shown in the following code:

```
GET {{base_address}}77
```

- Note that the response includes headers to control caching, as highlighted in the following output:

```
Response time: 89 ms Status code: OK (200)
Alt-Svc: h3=":5111"; ma=86400 Transfer-
Encoding: chunked
Vary: User-Agent
Cache-Control: public, max-age=5
Date: Fri, 09 Jun 2025 06:26:45 GMT
Server: Kestrel Content-Type: application/
json; charset=utf-8 Content-Length: 270
-----
Content: { "productId": 77, "productName":
"Original Frankfurter grüne Soße",
"supplierId": 12, "categoryId": 2,
"quantityPerUnit": "12 boxes",
"unitPrice": 85.0, "unitsInStock": 32,
"unitsOnOrder": 0, "reorderLevel": 15,
"discontinued": false, "category": null,
"orderDetails": [], "supplier": null }
```

- Close the browser and shut down the web server.

Good practice: Response caching should only be enabled for anonymous requests. Authenticated requests and responses should not be cached.

Caching is one of the best ways to improve the performance and scalability of your services. Next, we will learn how to improve a service's resilience when inevitable failures occur.

Fault tolerance with Polly

Polly is “a .NET resilience and transient-fault-handling library that allows developers to express policies such as *Retry*, *Circuit Breaker*, *Timeout*, *Bulkhead Isolation*, and *Fallback* in a fluent and thread-safe manner,” as stated on the official Polly GitHub repository, which can be found at the following link: <https://github.com/App-vNext/Polly>.

Transient faults are errors caused by temporary conditions, such as temporary service unavailability or network connectivity issues. It is essential to handle transient faults in distributed systems, or they can become almost unusable.

Understanding retry and circuit breaker patterns

The **Retry** pattern enables clients to automatically retry a failed action with the expectation that the fault will succeed if retried after a short delay. Be careful, because if you implement the Retry pattern naively, then it can make the problem worse!

For example, if you set a fixed time between retries, then all the clients who received a fault will attempt to retry at the same time, overloading the service. To avoid this issue, retries are often set with an exponentially increasing time between retries, or they might use jitter (aka randomizer) algorithms.

The **Circuit Breaker** pattern prevents calls when a threshold of faults is reached. In effect, it is a way for a service to detect if a fault is *not* transient, or not transient enough to keep retrying.

There is a nice summary table of resilience policies for Polly on its GitHub repository: <https://github.com/App-vNext/Polly#resilience-policies>. You can read about the circuit breaker policy fine-tuning best

practice at the following link: <https://devblogs.microsoft.com/dotnet/circuit-breaker-policy-finetuning-best-practice/>.

Defining and executing policies

In any type of .NET project that calls any unreliable code, you can reference the Polly package and then define a policy using the `Policy` class. Polly is not used in the unreliable code or service itself. It is used by any clients that call the code or service.

For example, you might need to call two methods that might throw arithmetic or custom exceptions, and you want to automatically retry up to three times, so you define a policy to handle this, as shown in the following code:

```
RetryPolicy policy = Policy
    .Handle<CustomException>().Or<ArithmeticException>()
    .Retry(3);
```

Then, you can use that policy to execute the methods, as shown in the following code:

```
policy.Execute(() => GetProducts());
policy.Execute(() => GetCustomers());
```

Each call to `Execute` gets its own counter for retries, so if the call to `GetProducts` needs two retries, the call to `GetCustomers` still has a full three retries of its own.

For unlimited retries, you can call the `RetryForever` method, but this is not recommended.

For asynchronous methods, there are matching asynchronous methods; for example, instead of `Retry`, use `RetryAsync`.

To execute some statements when a retry occurs, for example, to log information, the `Retry` method can be passed a callback method to call each time it needs to retry, as shown in the following code:

```
RetryPolicy policy = Policy
    .Handle<CustomException>().Or<ArithmeticException>()
    .Retry(3, onRetry: (exception, retryCount) =>
        { // Log the current retry count and exception
          information. });
```

Defining wait intervals between retries

Instead of immediately retrying after a fault, it is good practice to wait a moment before retrying.

For example, to wait and retry, as shown in the following code:

```
RetryPolicy policy = Policy
    .Handle<CustomException>().Or<ArithmeticException>()
    .WaitAndRetry(new[] { TimeSpan.FromSeconds(1),
        // 1 second between 1st and 2nd try.
        TimeSpan.FromSeconds(2), // 2 seconds between
        2nd and 3rd try.
        TimeSpan.FromSeconds(5) // 5
        seconds between 3rd and 4th try. });
```

Instead of hardcoded delay values, you can also define a function to generate them, as shown in the following code:

```
RetryPolicy policy = Policy
    .Handle<CustomException>().Or<ArithmeticException>()
```



```
.WaitAndRetry(3, retryAttempt =>
    TimeSpan.FromSeconds(Math.Pow(2,
        retryAttempt))); // 2 ^ 1 = 2 seconds then //
                        // 2 ^ 2 = 4 seconds then // 2 ^ 3 = 8 seconds
                        then
```

However, if we pass an array of fixed delays, even if they are calculated, imagine what happens when a fault occurs in a busy web service. All the clients receive an exception, they all wait for the first second, and they all attempt to recall the web service one second later. This causes floods that could make the situation worse!

Jittering is the idea of adding small amounts of randomization to time delays. There are many implementations that you can find online, and the best is built-in with an extra Polly package. We will use it to generate time delays in our example project.

Applying policies to HTTP clients

When calling a web service, it's a good practice to define an HTTP client factory and register it in a dependency services collection.

In this scenario, you will not call the methods that might throw an exception yourself. Instead, you must define a policy and then attach it to a registered HTTP client, so that it automatically follows that policy.

To do so, we will use an extension class named `HttpPolicyExtensions` to create policies specifically for common HTTP requests and failures, as shown in the following code:

```
AsyncRetryPolicy<HttpResponseMessage>
retryPolicy = HttpPolicyExtensions // Handle
network failures, 408 and 5xx status codes.
.HandleTransientHttpError() // Define the
policy using all the same options as before.
```

```
.RetryAsync(3);
```

To attach the policy to an HTTP client, call the `AddPolicyHandler` extension method after defining the factory. You will see how to do this in practice later in this section.

Adding random faults to the web service

First, let's add random faults to the web service:

1. In the `Northwind.WebApi.Controllers` project, in `ProductsController.cs`, in the `Get` action method that has a `name` parameter, add statements to randomly throw an exception two-thirds of the time, as highlighted in the following code:

```
// GET api/products/cha
[HttpGet("{name}")] public
IEnumerable<Product> Get(string name) { //
Works correctly 1 out of 3 times. if
    (Random.Shared.Next(1, 4) == 1) { return
        _db.Products.Where(p =>
            p.ProductName.Contains(name)); } // Throws
an exception at all other times. throw new
    Exception("Randomized fault."); }
```

2. Build the project.

Building an MVC project to call the faulty web service

Next, let's add to the existing ASP.NET Core MVC client that calls the randomly faulty web service endpoint. Initially, it will just receive the exception if the web service throws an exception. Later, we will add transient fault

handling using Polly:

1. In the `Northwind.WebApi.Client.Mvc` project, in `Program.cs`, before calling the `builder.Build()`, add statements to configure an HTTP client factory to call the controller-based web service, as shown in the following code:

```
builder.Services.AddHttpClient(name:
    "Northwind.WebApi.Controllers",
    configureClient: options => {
        options.BaseAddress = new("https://
            localhost:5111/");
        options.DefaultRequestHeaders.Accept.Add(
            new MediaTypeWithQualityHeaderValue(
                "application/json", 1.0)); });
```

2. In the `Controllers` folder, in `HomeController.cs`, add an asynchronous action method named `ProductsWithPolly`, which will use the HTTP client factory to request products whose name contains a value entered as an optional `name` parameter, in a custom MVC route, as shown in the following code:

```
[Route("home/productspolly/{name?}")]
public async Task<IActionResult>
ProductsWithPolly(string? name = "cha") {
    HomeProductsViewModel model = new();
    HttpClient client =
        _httpClientFactory.CreateClient( name:
            "Northwind.WebApi.Controllers");
    model.NameContains = name;
    model.BaseAddress = client.BaseAddress;
    HttpRequestMessage request = new( method:
        HttpMethod.Get, requestUri: $"api/
            products/{name}"); HttpResponseMessage
```

```

response = await
client.SendAsync(request); if
(response.IsSuccessStatusCode) {
model.Products = await response.Content
.ReadFromJsonAsync<IEnumerable<Product>>();
} else { model.Products =
Enumerable.Empty<Product>(); string
content = await
response.Content.ReadAsStringAsync(); //
Use the range operator .. to start from
zero and // go to the first carriage
return. string exceptionMessage =
content[..content.IndexOf("\r")];
model.ErrorMessage = string.Format("{0}:
{1}:", response.ReasonPhrase,
exceptionMessage); } return View(model); }

```

3. In the Views/Home folder, add a new file named ProductsWithPolly.cshtml. (The Visual Studio project item template is named **Razor View - Empty**. The Rider project item template is named **Razor MVC View**.)
4. In ProductsWithPolly.cshtml, modify its contents to output a table of products that match part of a product name entered in a textbox, as shown in the following markup:

```

@using Northwind.EntityModels @model
HomeProductsViewModel @{ ViewData["Title"]
= "Products using Polly"; } <div
class="text-center"> <h1
class="display-4">@ViewData["Title"]</h1>
<div class="alert alert-info"> <p> This
page calls a web service endpoint that
will randomly fail two out of three times.
It will use Polly to retry the call

```

```

automatically. </p> </div> @if (Model is
not null) { if (!
string.IsNullOrEmpty(Model.ErrorMessage))
{ <div class="alert alert-danger">
@Model.ErrorMessage </div> } <form
action="/home/productspolly"> <input
name="name" placeholder="Enter part of a
product name" value="@Model.NameContains"
/> <input type="submit" value="Get
Products" /> @if (!
string.IsNullOrEmpty(Model.NameContains))
{ <p> Searched for product names that
start with: <span class="badge bg-primary
rounded-pill"> @Model.NameContains</span>
</p> } </form> <div> @if (Model.Products
is not null) { <table class="table">
<thead> <tr> <th scope="col">Product
Name</th> </tr> </thead> <tbody> @if
(Model.Products.Any()) { @foreach (Product
p in Model.Products) { <tr> <td> <a href=
"@ (Model.BaseAddress)api/products/
@p.ProductId"> @p.ProductName</a> </td> </
tr> } } else { <tr><td>0 products found.</
td></tr> } </tbody> </table> } </div> } </
div>

```

5. In Views/Home, in Index.cshtml, add code to define a link to the Products using Polly page, as shown in the following markup:

```

<p><a href="home/productspolly">Search for
products by name using Polly</a></p>

```

6. Start the Northwind.WebApi.Controllers project, using the https profile without debugging.

7. Start the `Northwind.WebApi.Client.Mvc` project, using the `https` profile without debugging.

If you are using VS Code, then the web browser will not start automatically. Start Chrome, and then navigate to `https://localhost:5102`.

1. On the home page, click **Search for products by name using Polly**.
2. If the search works, you will see the successful results shown in *Figure 11.1*:

Products using Polly

This page calls a web service endpoint that will randomly fail two out of three times. It will use Polly to retry the call automatically.

Searched for: cha

Product Name

[Chai](#)

[Chang](#)

[Chartreuse verte](#)

Figure 11.1: A successful call to the faulty random web service

1. When it fails, you will see an error message, as shown in *Figure 11.2*:

Products using Polly

This page calls a web service endpoint that will randomly fail two out of three times. It will use Polly to retry the call automatically.

Internal Server Error: System.Exception: Randomized fault.

Searched for: man

Product Name

0 products found.

Figure 11.2: A failed call to the faulty random web service

1. In the command prompt or terminal, when a fault occurs, you will see the exception, as shown in the following partial output:

```
fail:
Microsoft.AspNetCore.Diagnostics.DeveloperExceptionPageMiddleware
An unhandled exception has occurred while
executing the request. System.Exception:
Randomized fault.
```

2. Enter different partial names and click **Get Products** until you have seen both a successful search and a failed search.
3. Close the browsers and shut down the web servers.

Implementing the Retry pattern for transient fault handling

Now that we have a web service and MVC client with random faults, let's add transient fault handling by using the Retry pattern:

1. In the `Northwind.WebApi.Client.Mvc` project file, globally and statically import the `System.Console` class, and add a package reference for the Microsoft package to integrate Polly with ASP.NET Core (which has a dependency on the `Polly` package), and for a library to add jittering to retry time spans, as shown in the following markup:

```
<ItemGroup> <Using
Include="System.Console" Static="true"/>
</ItemGroup> <ItemGroup> <PackageReference
Include="Microsoft.Extensions.Http.Polly"
/> <PackageReference
Include="Polly.Contrib.WaitAndRetry" /> </
```

```
ItemGroup>
```

2. Build the `Northwind.WebApi.Client.Mvc` project to restore packages.
3. In `Program.cs`, import common Polly namespaces to work with ASP.NET Core, as shown in the following code:

```
using Polly; // To use
AddTransientHttpErrorPolicy method. using
Polly.Contrib.WaitAndRetry; // To use
Backoff. using Polly.Extensions.Http; //
To use HttpPolicyExtensions. using
Polly.Retry; // To use AsyncRetryPolicy<T>
```

4. In `Program.cs`, before the statement to add an HTTP client to services, add statements to generate five jittered and exponentially increasing time-span values, output them to the console, use them to define an asynchronous wait and retry policy, and then add the retry policy to the HTTP client factory, as highlighted in the following code:

```
// Create five jittered delays, starting
with about 1 second. IEnumerable<TimeSpan>
delays =
Backoff.DecorrelatedJitterBackoffV2(
medianFirstRetryDelay:
TimeSpan.FromSeconds(1), retryCount: 5);
WriteLine("Jittered delays for Polly
retries:"); foreach (TimeSpan item in
delays) { WriteLine($"
{item.TotalSeconds:N2} seconds."); }
AsyncRetryPolicy<HttpResponseMessage>
retryPolicy = HttpPolicyExtensions //
```


Handle network failures, 408 and 5xx status codes.

```
.HandleTransientHttpError().WaitAndRetryAsync(del  
builder.Services.AddHttpClient(name:  
"Northwind.WebApi.Controllers",  
configureClient: options => {  
options.BaseAddress = new("https://  
localhost:5111/");  
options.DefaultRequestHeaders.Accept.Add(  
new MediaTypeWithQualityHeaderValue(  
"application/json", 1.0)); })  
.AddPolicyHandler(retryPolicy);
```

5. If your database server is not running (for example, because you are hosting it in Docker, a virtual machine, or in the cloud), then make sure to start it.
6. Start the `Northwind.WebApi.Controllers` project, using the `https` profile without debugging.
7. Start the `Northwind.WebApi.Client.Mvc` project, using the `https` profile without debugging.
8. In the command prompt or terminal for the MVC project, note the jittered time spans, as shown in the following output:

```
Jittered delays for Polly retries: 1.38  
seconds. 0.15 seconds. 2.65 seconds. 3.06  
seconds. 6.46 seconds.
```

Your five delays will be different, but they should start at about 1 second and increase from there.

1. Arrange the web service command prompt or terminal and the MVC

website browser so that you can see both side by side.

2. On the home page, click **Search for products by name using Polly**.
3. Note that the MVC website might have to make multiple requests before showing the page, which will take up to about 15 seconds. For example, when I ran my projects, the MVC web site made four requests that failed before succeeding on the fifth attempt. You will see the exceptions logged in the web service output.
4. Enter a partial product name, click **Get Products**, and note that the web page will likely appear successfully again, even if one or more requests must be made beforehand.
5. It is possible that you could exceed the maximum of five requests, in which case you will see the error message as before.

Microsoft has created their own packages that wrap Polly to make it even easier to use. They are the `Microsoft.Extensions.Http.Resilience` and `Microsoft.Extensions.Resilience` packages. You can learn about this at the following link: <https://devblogs.microsoft.com/dotnet/building-resilient-cloud-services-with-dotnet-8/>.

Now that you've seen two techniques that improve services, caching, and handling transient faults, let's look at a third powerful technique, queuing.

Queuing with RabbitMQ

Queuing can improve the scalability of your service, just as it can in the physical world. When too many clients all need to call a service at once, we can use a queue to smooth out the load.

There are many queuing systems available for all the major development

platforms. One of the most popular is RabbitMQ. It implements the **Advanced Message Queuing Protocol (AMQP)**.

With AMQP, messages are published to exchanges, which then distribute message copies to queues using rules named bindings. Then a broker can deliver the messages to consumers subscribed to a queue (sometimes called a topic), or a consumer can read from a queue when they want.

Since networks and systems often fail, AMQP uses message acknowledgements to tell the broker when a consumer has successfully processed a message, and only then does the broker remove the message from the queue.

RabbitMQ supports four types of exchanges:

- **Direct:** A direct exchange delivers messages based on a message routing key. Multiple queues can be bound to the exchange, but messages are only delivered to a queue if they have a matching routing key. They are mostly used for unicast messages. The default (empty name) exchange is a direct exchange. It is pre-bound with a routing key that is the same name as the queue. This is the type we will use in this book.
- **Fanout:** A fanout exchange delivers messages to all queues that are bound to it, and the routing key is ignored. These are good for broadcasting messages.
- **Topic:** A topic exchange delivers messages based on a routing key and criteria defined in the binding between the exchange and a queue. They are used for the publish/subscribe pattern, where there are many consumers but they want to receive different messages, based on factors like geographic location, registered interests, and so on.
- **Headers:** A headers exchange delivers messages based on multiple attributes in a message header instead of a routing key.

You can better understand RabbitMQ by using RabbitMQ Simulator. You can access it at the following link: <https://>

tryrabbitmq.com/.

The RabbitMQ API uses the following types:

- `IConnection`: This represents an AMQP connection.
- `ConnectionFactory`: This creates `IConnection` instances. It has default values for common properties designed to work with the Docker image. For example, `Username` is `guest`, `Password` is `guest`, `VirtualHost` is `/`, `HostName` is `localhost`, and `Port` is `5672`.
- `IModel`: This represents the AMQP channel and has methods to perform common tasks, like declaring a queue with `QueueDeclare` or sending a message using `BasicPublish`.
- `IBasicConsumer`: This represents a message consumer.
- `EventBasicConsumer`: This is an implementation of a message consumer that integrates with the .NET event system, making it easy for a client app to process a message as soon as it is sent and received.

Good practice: Queuing systems can get complicated fast. In this book, we will cover the basics, but if you decide to implement any queuing system in production, then you will want to learn much more about how to implement them deeply.

You can install RabbitMQ locally on your computer, but for maximum ease of use, I recommend using a Docker image.

To install RabbitMQ on your computer, read the instructions at the following link for your operating system: <https://www.rabbitmq.com/download.html>.

Setting up RabbitMQ using Docker

The Docker image we will use has RabbitMQ version 3.13 and is designed to be used as a throwaway container, where you simply start the container and your project can start using it with the default configuration.

You can read more about the Docker image at the following link: https://registry.hub.docker.com/_/rabbitmq/.

Let's get started with RabbitMQ in a Docker container:

1. Start **Docker**.
2. At the command prompt or terminal, pull down the latest container image for RabbitMQ on Docker and run it, opening ports 5672 and 15672 to the container, which are used by default by AMQP, as shown in the following command:

```
docker run -it --rm --name rabbitmq -p
5672:5672 -p 15672:15672 rabbitmq:3.13.7-
management
```

3. Note that the first time you run this command, the RabbitMQ image will not be found on your local computer, as shown in the following output:

```
Unable to find image 'rabbitmq:3.13.7-
management' locally
```

4. Note that the image will then be downloaded automatically, as shown in the following output:

```
3.13.7-management: Pulling from library/  
rabbitmq 99803d4b97f3: Pull complete  
8fb904ec525a: Pull complete ba4d114a87c0:  
Pull complete c869b027f1e1: Pull complete  
729c8b3166a8: Pull complete 7de098b90abf:  
Pull complete 4f206ad5199f: Pull complete  
1f40437d763f: Pull complete f4cbf27a2d68:  
Pull complete 5a4db5ea38b2: Pull complete  
99886074092c: Pull complete Digest:  
sha256:da98d468cf2236171da94e34953619ddd01b5dk  
Status: Downloaded newer image for  
rabbitmq:3.12-management
```

5. Note that RabbitMQ runs on Erlang, and its copyright and license information are displayed when the container starts up, as shown in the following output:

```
2025-12-11 14:03:22.785019+00:00 [info]  
<0.254.0> Starting RabbitMQ 3.13.7 on  
Erlang 26.2.5.16 [jit] 2025-12-11  
14:03:22.785019+00:00 [info] <0.254.0>  
Licensed under the MPL 2.0. Website:  
https://rabbitmq.com ## ## RabbitMQ 3.13.7  
## ## ##### Copyright (c) 2007-2024  
Broadcom Inc. or its affiliates. ##### ##  
##### Licensed under the MPL 2.0.  
Website: https://rabbitmq.com
```

6. Note the RabbitMQ service listens on port 5672 and has started five plugins, as shown in the following output:

```
2025-12-12 17:45:38.634300+00:00 [info]  
<0.754.0> Management plugin: HTTP (non-
```

```
TLS) listener started on port 15672
2025-12-12 17:45:38.634429+00:00 [info]
<0.784.0> Statistics database started.
2025-12-12 17:45:38.634524+00:00 [info]
<0.783.0> Starting worker pool
'management_worker_pool' with 3 processes
in it 2025-12-12 17:45:38.643016+00:00
[info] <0.802.0> Prometheus metrics: HTTP
(non-TLS) listener started on port 15692
2025-12-12 17:45:38.643154+00:00 [info]
<0.688.0> Ready to start client connection
listeners 2025-12-12 17:45:38.644842+00:00
[info] <0.846.0> started TCP listener on
[::]:5672 completed with 5 plugins.
2025-12-12 17:45:38.751167+00:00 [info]
<0.688.0> Server startup complete; 5
plugins started. 2025-12-12
17:45:38.751167+00:00 [info] <0.688.0> *
rabbitmq_prometheus 2025-12-12
17:45:38.751167+00:00 [info] <0.688.0> *
rabbitmq_federation 2025-12-12
17:45:38.751167+00:00 [info] <0.688.0> *
rabbitmq_management 2025-12-12
17:45:38.751167+00:00 [info] <0.688.0> *
rabbitmq_management_agent 2025-12-12
17:45:38.751167+00:00 [info] <0.688.0> *
rabbitmq_web_dispatch 2025-12-12
17:45:38.894997+00:00 [info] <0.9.0> Time
to start RabbitMQ: 7530 ms
```

7. Leave the command prompt or terminal running.
8. Optionally, in **Docker Desktop**, note that a container for RabbitMQ runs and listens on ports 5672 (the actual queue service) and 15672 (its management service), as shown in *Figure 11.3*:

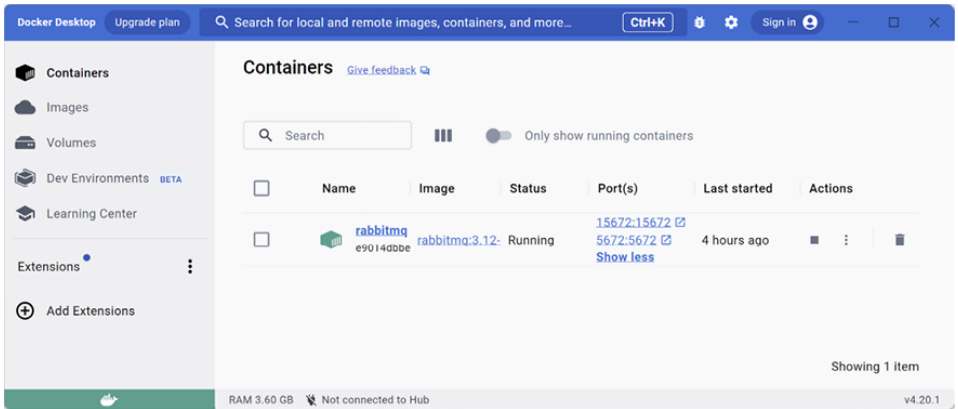


Figure 11.3: RabbitMQ running in a Docker container

Sending messages to a queue using an MVC website

Now that we have the RabbitMQ system running, we can add the RabbitMQ client package to the MVC website project so that it can send messages to a queue.

But first, let's create a class library to define models we will use with the queue:

1. Use your preferred code editor to create a new class library project, as defined in the following list:
 - Project template: **Class Library** / `classlib`
 - Solution file and folder: `ModernServices`
 - Project file and folder: `Northwind.Queue.Models`
5. Add a project reference to the Northwind entity models project, as shown in the following markup:

```
<ItemGroup> <ProjectReference Include="..\
  \..\ModernApps\ Northwind.EntityModels
  \Northwind.EntityModels.csproj" /> </
ItemGroup>
```


6. At the command prompt or terminal, build the project to make sure the entity model class library projects outside the current solution are properly compiled, as shown in the following command:

```
dotnet build
```

7. Delete the file named `Class1.cs`.
8. Add a new file named `ProductQueueMessage.cs`.
9. In `ProductQueueMessage.cs`, define a class that will represent a message in a queue with a simple plain text property and a complex `Product` entity model type as a second property, as shown in the following code:

```
using Northwind.EntityModels; // To use
Product. namespace Northwind.Queue.Models;
public class ProductQueueMessage { public
string? Text { get; set; } public Product
Product { get; set; } = null!; }
```

10. In the `Northwind.WebApi.Client.Mvc` project file, add a reference to the queue models project so that we can use the `ProductQueueMessage` class, as shown in the following markup:

```
<ItemGroup> <ProjectReference Include= "..
\Northwind.Queue.Models
\Northwind.Queue.Models.csproj" /> </
ItemGroup>
```

11. In the `Northwind.WebApi.Client.Mvc` project file, add a package reference for creating RabbitMQ clients, as shown in the following markup:

```
<PackageReference  
  Include="RabbitMQ.Client" />
```

12. Build the `Northwind.WebApi.Client.Mvc` project.
13. In the `Models` folder, add a new class file named `HomeSendMessageViewModel.cs`.
14. Define a class to represent the information that needs to be displayed in a view for sending a message including a couple of properties for showing message to the visitor when a message is successfully sent and when an error occurs, as shown in the following code:

```
using Northwind.Queue.Models; // To use  
ProductQueueMessage. namespace  
Northwind.WebApi.Client.Mvc.Models; public  
class HomeSendMessageViewModel { public  
  string? Info { get; set; } public string?  
  Error { get; set; } public  
  ProductQueueMessage? Message { get; set; }  
}
```

15. In `Views\Home`, in `Index.cshtml`, add a link to a page that will let the visitor send a message to a queue, as shown in the following markup:

```
<p><a href="home/sendmessage">Send a  
message</a></p>
```

16. In `HomeController.cs`, import namespaces to work with `RabbitMQ` and to serialize `JSON`, as shown in the following code:

```
using RabbitMQ.Client; // To use  
ConnectionFactory and so on. using
```

```
System.Text.Json; // To use
JsonSerializer.
```

17. In `HomeController.cs`, add statements to define an action method that responds to a GET request by showing a web form to send a message, as shown in the following code:

```
public IActionResult SendMessage() {
    return View(); }
```

18. In `HomeController.cs`, add statements to define an action method that responds to a POST request by sending a message from information in the form, as shown in the following code:

```
// POST: home/sendmessage // Body:
message=Hello&productId=1 [HttpPost]
public async Task<IActionResult>
SendMessage( string? message, int?
productId) { HomeSendMessageViewModel
model = new(); model.Message = new(); if
(message is null || productId is null) {
model.Error = "Please enter a message and
a product ID."; return View(model); }
model.Message.Text = message;
model.Message.Product = new() { ProductId
= productId.Value }; HttpClient client =
_httpClientFactory.CreateClient( name:
"Northwind.WebApi.Controllers");
HttpRequestMessage request = new( method:
HttpMethod.Get, requestUri: $"api/
products/{productId}");
HttpResponseMessage response = await
client.SendAsync(request); if
```

```

(response.IsSuccessStatusCode) { Product?
product = await response.Content
.ReadFromJsonAsync<Product>(); if (product
is not null) { model.Message.Product =
product; } } // Create a RabbitMQ factory.
ConnectionFactory factory = new() {
HostName = "localhost" }; using
IConnection connection = await factory
.CreateConnectionAsync(); using IChannel
channel = await
connection.CreateChannelAsync(); string
queueNameAndRoutingKey = "product"; // If
the queue does not exist, it will be
created. // If the Docker container is
restarted, the queue will be lost. // The
queue can be shared with multiple
consumers. // The queue will not be
deleted when the last message is consumer.
await channel.QueueDeclareAsync( queue:
queueNameAndRoutingKey, durable: false,
exclusive: false, autoDelete: false,
arguments: null); byte[] body =
JsonSerializer.SerializeToUtf8Bytes(model.Message)
// The exchange is empty because we are
using the default exchange.
channel.BasicPublish( exchange:
string.Empty, routingKey:
queueNameAndRoutingKey, body: body);
model.Info = "Message sent to queue
successfully."; return View(model); }

```

19. In Views\Home, add a new empty Razor View named
SendMessage.cshtml.

20. Define a web page with a form to send a message, as shown in the

following markup:

```
@model HomeSendMessageViewModel @{
    ViewData["Title"] = "Send a Message"; }
<div class="text-center"> <h1
class="display-4">@ViewData["Title"]</h1>
@if (Model is not null) { if (Model.Error
is not null) { <div class="alert alert-
danger"> <h2>Error</h2> <p>@Model.Error</
p> </div> } if (Model.Info is not null) {
<div class="alert alert-info">
<h2>Information</h2> <p>@Model.Info</p> </
div> } } <form asp-controller="Home" asp-
action="SendMessage" method="post"> <div>
<label for="message">Message</label>
<input id="message" name="message" /> </
div> <div> <label for="productId">Product
ID</label> <input id="productId"
name="productId" /> </div> <input
type="submit" value="Send" />" </form> </
div>
```

Consuming messages from a queue using a console app

Finally, we can create a console app that will process messages from the queue:

1. Use your preferred code editor to create a new console app project, as defined in the following list:
 - Project template: **Console App** / console
 - Solution file and folder: ModernServices
 - Project file and folder:
Northwind.Queue.Consumer

5. Treat warnings as errors, add a package reference for RabbitMQ, add project references to the Northwind entity models project and the queue message models project, and statically and globally import the `System.Console` class, as shown in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup> <OutputType>Exe</
  OutputType> <TargetFramework>net10.0</
  TargetFramework> <ImplicitUsings>enable</
  ImplicitUsings> <Nullable>enable</
  Nullable> <TreatWarningsAsErrors>true</
  TreatWarningsAsErrors> </PropertyGroup>
  <ItemGroup> <PackageReference
  Include="RabbitMQ.Client" /> </ItemGroup>
  <ItemGroup> <ProjectReference Include="..
  \..\ModernApps\ Northwind.EntityModels
  \Northwind.EntityModels.csproj" />
  <ProjectReference Include="..
  \Northwind.Queue.Models\
  Northwind.Queue.Models.csproj" /> </
  ItemGroup> <ItemGroup> <Using
  Include="System.Console" Static="true" />
  </ItemGroup> </Project>
```

6. At the command prompt or terminal, build the project, as shown in the following command:

```
dotnet build
```

7. In `Program.cs`, delete any existing statements, and then add statements to read messages from the product queue, as shown in the following code:

```

using Northwind.Queue.Models; // To use
ProductQueueMessage. using
RabbitMQ.Client; // To use
ConnectionFactory. using
RabbitMQ.Client.Events; // To use
EventingBasicConsumer. using
System.Text.Json; // To use
JsonSerializer. string queueName =
"product"; ConnectionFactory factory = new
() { HostName = "localhost" }; using
IConnection connection = await
factory.CreateConnectionAsync(); using
IChannel channel = await
connection.CreateChannelAsync();
WriteLine("Declaring queue...");
QueueDeclareOk response = await
channel.QueueDeclareAsync( queue:
queueName, durable: false, exclusive:
false, autoDelete: false, arguments:
null); WriteLine("Queue name:
{response.QueueName}, Message count: {
response.MessageCount}, Consumer count:
{response.ConsumerCount}.");
WriteLine("Waiting for messages...");
AsyncEventingBasicConsumer consumer =
new(channel); consumer.Received += async
(model, args) => { byte[] body =
args.Body.ToArray(); ProductQueueMessage?
message = JsonSerializer
.Deserialize<ProductQueueMessage>(body);
if (message is not null) {
WriteLine("Received product. Id:
{message.Product.ProductId }, Name: {
message.Product.ProductName}, Message: {
message.Text}"); } else {

```

```
WriteLine($"Received unknown:
{args.Body.ToArray()}."); } }]; // Start
consuming as messages arrive in the queue.
await channel.BasicConsumeAsync(queue:
queueName, autoAck: true, consumer:
consumer); WriteLine(">>> Press Enter to
stop consuming and quit. <<<");
ReadLine();
```

8. If your database server is not running (for example, because you are hosting it in Docker, a virtual machine, or in the cloud), then make sure to start it.
9. Start the `Northwind.WebApi.Controllers` project, using the `https` profile without debugging.
10. Start the `Northwind.WebApi.Client.Mvc` project, using the `https` profile without debugging.
11. Start the `Northwind.Queue.Consumer` console app project with or without debugging.

Optionally, you can configure your solution to start up all three projects at once, as shown for Visual Studio in *Figure 11.4*:

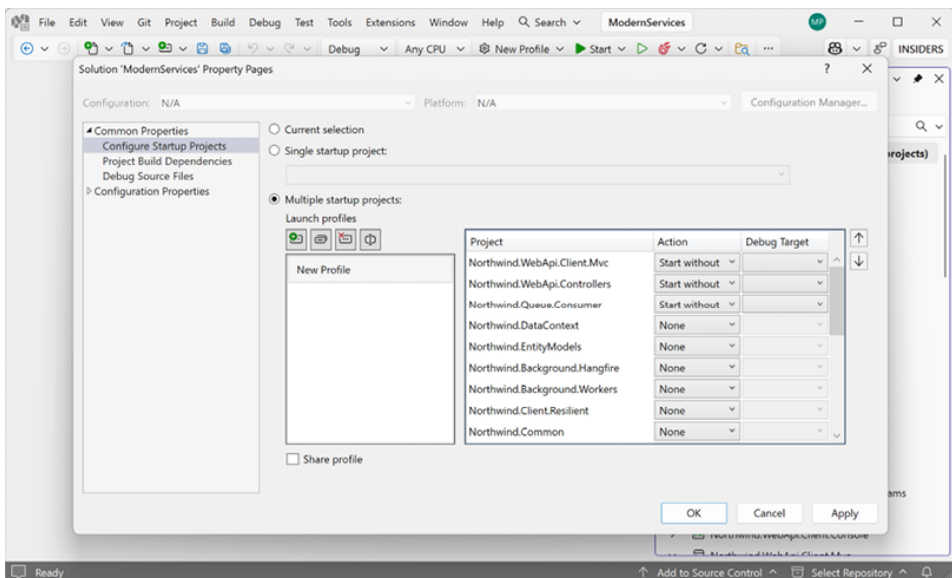


Figure 11.4: Configuring three startup projects to test message queues

1. Arrange the console app and the MVC web page so that you can see both.
2. On the home page, click **Send a message**, and enter a simple text message and a valid product ID (1 to 77), as shown in Figure 11.5:

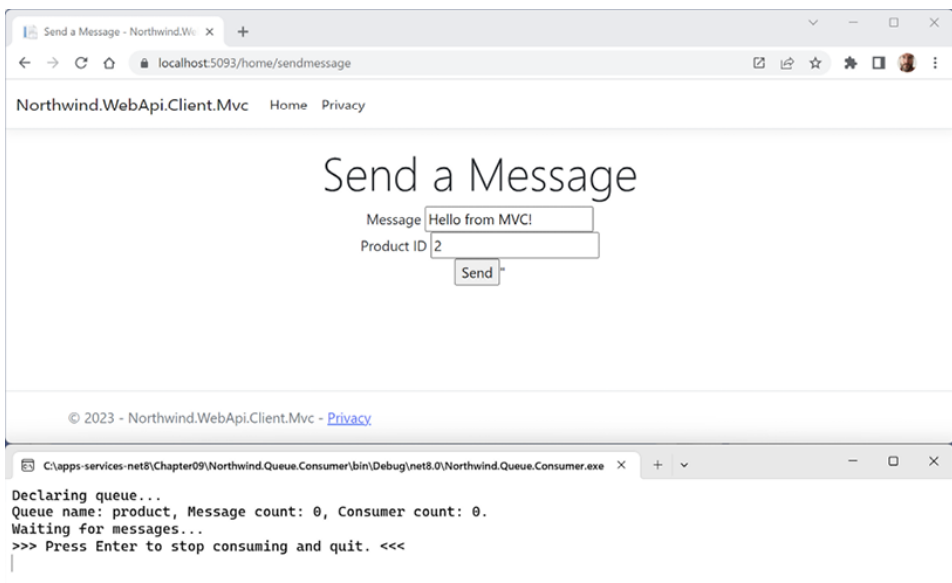


Figure 11.5: An ASP.NET Core MVC website sending a message to a queue

1. Click **Send**, and note the message that appears in the console app, as shown in *Figure 11.6*:

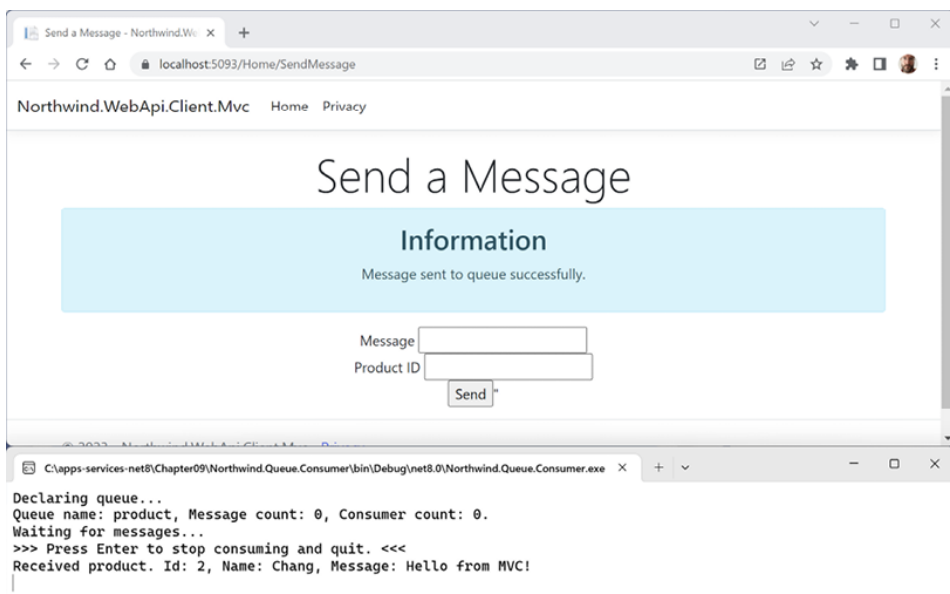


Figure 11.6: A console app consuming a message from the queue

1. In the command prompt or terminal for Docker, press **Ctrl + C** to shut down the container.
2. In **Docker Desktop**, note the container is gone from the list, but the image remains for quicker use next time.

You can read more about using RabbitMQ with .NET at the following link: <https://www.rabbitmq.com/dotnet-api-guide.html>.

The combination of caching, queuing, and handling transient faults goes a long way toward making your services more resilient, scalable, and performant. In the last section of this chapter, we will look at long-running background

services.

Implementing long-running background services

It is common to need long-running background services to perform operations like:

- Performing a task on a regular timed schedule.
- Processing queued messages.
- Performing intense work like building AI and ML models or processing video and images.

In the distant past, on the Windows operating system, to have some code running in the background meant building a **Windows Service**. For example, the database engine of SQL Server is implemented as a Windows Service. With the move to cross-platform, .NET needs a cross-platform solution to run code in the background.

Background services often do not have a user interface, although they might provide one for the configuration and management of the service.

Building a worker service

Now, let's build a worker service project so that we can see how we would host a long-running background service:

1. Use your preferred code editor to add a new project, as defined in the following list:
 - Project template: **Worker Service** / `worker`
 - Solution file and folder: `ModernServices`
 - Project file and folder:
`Northwind.Background.Workers`
 - **Enable container support**: Cleared

- **Do not use top-level statements:** Cleared
- **Enable native AOT publish:** Cleared

8. In the `Northwind.Background.Workers` project file, note that the .NET SDK is `Microsoft.NET.Sdk.Worker`, and then make the following changes, as highlighted in the following markup:

- Treat warnings as errors.
- Add a package reference for RabbitMQ.
- Add references to the entity models and queue models projects:

```
<Project Sdk="Microsoft.NET.Sdk.Worker">
  <PropertyGroup> <TargetFramework>net10.0</
TargetFramework> <Nullable>enable</
Nullable> <ImplicitUsings>enable</
ImplicitUsings> <UserSecretsId>dotnet-
Northwind.Background.Workers-66434cdf-0fdd-4993-
a399-ec9581b4b914</UserSecretsId>
  <TreatWarningsAsErrors>true</
TreatWarningsAsErrors> </PropertyGroup>
  <ItemGroup> <PackageReference
Include="Microsoft.Extensions.Hosting" />
  <PackageReference
Include="RabbitMQ.Client" /> </ItemGroup>
  <ItemGroup> <ProjectReference Include="..
..\ModernApps\ Northwind.EntityModels
\Northwind.EntityModels.csproj" />
  <ProjectReference Include="..
\Northwind.Queue.Models\
Northwind.Queue.Models.csproj" /> </
ItemGroup> </Project>
```

12. Build the `Northwind.Background.Workers` project.

13. In `Program.cs`, note that the initialization statements are like an

ASP.NET Core project, and that it registers a hosted service named `Worker` and then runs the host, as shown in the following code:

```
using Northwind.Background.Workers; var
builder =
Host.CreateApplicationBuilder(args);
builder.Services.AddHostedService<Worker>();
var host = builder.Build(); host.Run();
```

14. In `Worker.cs`, note that the `Worker` class inherits from `BackgroundService` and implements its `ExecuteAsync` method by looping until a cancellation is requested, logging the current date/time, and then pausing for one second, as shown in the following code:

```
namespace Northwind.Background.Workers {
public class Worker : BackgroundService {
private readonly ILogger<Worker> _logger;
public Worker(ILogger<Worker> logger) {
_logger = logger; } protected override
async Task ExecuteAsync( CancellationToken
stoppingToken) { while (!
stoppingToken.IsCancellationRequested) {
_logger.LogInformation("Worker running at:
{time}", DateTimeOffset.Now); await
Task.Delay(1000, stoppingToken); } } }
```

15. Start the project without debugging, note the current time is output once per second, and then press `Ctrl + C` to shut down the worker service, as shown in the following output:

```
info: Northwind.Queue.Worker.Worker[0]
Worker running at: 12/12/2025 08:25:02
```

```
+01:00 info: Microsoft.Hosting.Lifetime[0]
Application started. Press Ctrl+C to shut
down. info: Microsoft.Hosting.Lifetime[0]
Hosting environment: Development info:
Microsoft.Hosting.Lifetime[0] Content root
path: C:\apps-services-
net10\ModernServices
\Northwind.Queue.Worker info:
Northwind.Queue.Worker.Worker[0] Worker
running at: 12/12/2025 08:25:03 +01:00
info: Northwind.Queue.Worker.Worker[0]
Worker running at: 12/12/2025 08:25:04
+01:00 info:
Northwind.Queue.Worker.Worker[0] Worker
running at: 12/12/2025 08:25:05 +01:00
info: Microsoft.Hosting.Lifetime[0]
Application is shutting down...
```

Processing queued messages using a worker service

Now, we can do some useful work, like reading messages from a RabbitMQ queue:

1. Rename `Worker.cs` to `QueueWorker.cs` and the `Worker` class to `QueueWorker`.
2. In `Program.cs`, change the hosted service class name from `Worker` to `QueueWorker`, as shown in the following code:

```
builder.Services.AddHostedService<QueueWorker>();
```

3. In `QueueWorker.cs`, import namespaces to work with RabbitMQ

queues and implement a queue processor, as highlighted in the following code:

```
using Northwind.Queue.Models; // To use
ProductQueueMessage. using
RabbitMQ.Client; // To use
ConnectionFactory. using
RabbitMQ.Client.Events; // To use
AsyncEventingBasicConsumer. using
System.Text.Json; // To use
JsonSerializer. namespace
Northwind.Background.Workers; public class
QueueWorker : BackgroundService { private
readonly ILogger<QueueWorker> _logger; //
RabbitMQ objects. private
ConnectionFactory? _factory; private
IConnection? _connection; private
IChannel? _channel; private
AsyncEventingBasicConsumer? _consumer;
private const string
queueNameAndRoutingKey = "product"; public
QueueWorker(ILogger<QueueWorker> logger) {
_logger = logger; } public override async
Task StartAsync( CancellationToken
cancellationToken) { _factory = new() {
HostName = "localhost" }; _connection =
await _factory.CreateConnectionAsync();
_channel = await
_connection.CreateChannelAsync();
_consumer = new(_channel); await
_channel.QueueDeclareAsync( queue:
queueNameAndRoutingKey, durable: false,
exclusive: false, autoDelete: false,
arguments: null); _consumer =
new(_channel); _consumer.ReceivedAsync +=
```

```

async (model, args) => { byte[] body =
args.Body.ToArray(); ProductQueueMessage?
message = JsonSerializer
.Deserialize<ProductQueueMessage>(body);
if (message is not null) {
_logger.LogInformation($"Received product.
Id: { message.Product.ProductId}, Name:
{message.Product .ProductName}, Message:
{message.Text}"); } else {
_logger.LogInformation("Received unknown:
{0}.", args.Body.ToArray()); } }; // Start
consuming as messages arrive in the queue.
await _channel.BasicConsumeAsync( queue:
queueNameAndRoutingKey, autoAck: true,
consumer: _consumer); } protected override
async Task ExecuteAsync( CancellationToken
stoppingToken) { while (!
stoppingToken.IsCancellationRequested) {
_logger.LogInformation("Worker running at:
{time}", DateTimeOffset.Now); await
Task.Delay(3000, stoppingToken); } } }

```

4. Start the RabbitMQ container, as shown in the following command:

```

docker run -it --rm --name rabbitmq -p
5672:5672 -p 15672:15672 rabbitmq:3.13.7-
management

```

5. Wait for the messages to say it is ready for clients to connect on port 5672, as shown in the following output:

```

2025-12-12 08:50:16.591574+00:00 [info]
<0.599.0> Ready to start client connection

```



```
listeners 2025-12-12 08:50:16.593090+00:00  
[info] <0.744.0> started TCP listener on  
[::]:5672
```

6. Leave the command prompt or terminal running.
7. If your database server is not running (for example, because you are hosting it in Docker, a virtual machine, or in the cloud), then make sure to start it.
8. Start the `Northwind.WebApi.Controllers` project without debugging so that we can query for products in the Northwind database.
9. Start the `Northwind.WebApi.Client.Mvc` project without debugging so that we can send messages to the RabbitMQ queue.
10. In the MVC website, click **Send a message**, and then enter a message of apples and a product ID of 1.
11. Repeat for bananas and 2, and cherries and 3.
12. Start the `Northwind.Background.Workers` project without debugging, and note the three messages are processed from the queue, as shown in the following output:

```
info:  
Northwind.Background.Workers.QueueWorker[0]  
Queue product is waiting for messages.  
info:  
Northwind.Background.Workers.QueueWorker[0]  
Worker running at: 12/12/2025 09:58:59  
+01:00 info: Microsoft.Hosting.Lifetime[0]  
Application started. Press Ctrl+C to shut  
down. info: Microsoft.Hosting.Lifetime[0]  
Hosting environment: Development info:  
Microsoft.Hosting.Lifetime[0] Content root  
path: C:\apps-services-  
net10\ModernServices
```

```
\Northwind.Queue.Worker info:
Northwind.Background.Workers.QueueWorker[0]
Received product. Id: 1, Name: Chai,
Message: apples info:
Northwind.Background.Workers.QueueWorker[0]
Received product. Id: 2, Name: Chang,
Message: bananas info:
Northwind.Background.Workers.QueueWorker
[0] Received product. Id: 3, Name: Aniseed
Syrup, Message: cherries
```

Executing code on a timed schedule

Another common use of worker services is to implement timed events. A timer-based background service can use the `System.Threading.Timer` class, which triggers the `DoWork` method.

Let's add another service to the background worker project:

1. In the `Northwind.Background.Workers` project, add a new class named `TimerWorker.cs`.
2. Modify the class, as shown in the following code:

```
namespace Northwind.Background.Workers;
public class TimerWorker : IHostedService,
    IAsyncDisposable { private readonly
    ILogger<TimerWorker> _logger; private int
    _executionCount = 0; private Timer?
    _timer; private int _seconds = 5; public
    TimerWorker(ILogger<TimerWorker> logger) {
    _logger = logger; } private void
    DoWork(object? state) { int count =
    Interlocked.Increment(ref
    _executionCount); _logger.LogInformation(
```

```

"{0} is working, execution count:
{1: #,0}", nameof(TimerWorker), count); }
public Task StartAsync(CancellationToken
cancellationToken) {
_logger.LogInformation("{0} is running.",
nameof(TimerWorker)); _timer = new
Timer(callback: DoWork, state: null,
dueTime: TimeSpan.Zero, period:
TimeSpan.FromSeconds(_seconds)); return
Task.CompletedTask; } public Task
StopAsync(CancellationToken
cancellationToken) {
_logger.LogInformation("{0} is stopping.",
nameof(TimerWorker));
_timer?.Change(dueTime: Timeout.Infinite,
period: 0); return Task.CompletedTask; }
public async ValueTask DisposeAsync() { if
(_timer is IAsyncDisposable asyncTimer) {
await asyncTimer.DisposeAsync(); } _timer
= null; } }

```

3. In `Program.cs`, add a statement to register the timer worker service, as shown in the following code:

```
builder.Services.AddHostedService<TimerWorker>();
```

4. Start the `Northwind.Background.Workers` project without debugging, and note the initialization of both workers, as shown in the following output:

```

info:
Northwind.Background.Workers.QueueWorker[0]
Worker running at: 12/12/2025 12:58:25

```

```
+01:00 info:
Northwind.Background.Workers.TimerWorker[0]
TimerWorker is running. info:
Microsoft.Hosting.Lifetime[0] Application
started. Press Ctrl+C to shut down. info:
Microsoft.Hosting.Lifetime[0] Hosting
environment: Development info:
Microsoft.Hosting.Lifetime[0] Content root
path: C:\apps-services-
net10\ModernServices
\Northwind.Background.Workers
```

5. Leave the background workers running for at least 10 seconds, and note the queue worker writes to the log every second and the timer worker writes to the log every five seconds, as shown in the following output:

```
info:
Northwind.Background.Workers.TimerWorker[0]
TimerWorker is working, execution count: 1
info:
Northwind.Background.Workers.QueueWorker[0]
Worker running at: 06/12/2023 12:58:26
+01:00 info:
Northwind.Background.Workers.QueueWorker[0]
Worker running at: 06/12/2023 12:58:27
+01:00 info:
Northwind.Background.Workers.QueueWorker[0]
Worker running at: 06/12/2023 12:58:28
+01:00 info:
Northwind.Background.Workers.QueueWorker[0]
Worker running at: 06/12/2023 12:58:29
+01:00 info:
Northwind.Background.Workers.TimerWorker[0]
TimerWorker is working, execution count: 2
```

6. Press *Ctrl + C* to shut down the background workers, and note the clean shutdown of the timer worker, as shown in the following output:

```
info: Microsoft.Hosting.Lifetime[0]  
Application is shutting down... info:  
Northwind.Background.Workers.TimerWorker[0]  
TimerWorker is stopping.
```

If we wanted to use the timer background service to have more flexibility, instead of running it at a regular interval like five seconds, we could have it run a scheduled task check every second, and only if a task has reached its scheduled time does it then run that task. We need somewhere to define tasks and when they are scheduled to run. Although you can build this infrastructure yourself, it is easier to use a third-party library like **Hangfire**.

Building a website to host Hangfire

Hangfire is open source and free for commercial use. It supports the following patterns of use:

- **Fire-and-forget:** Jobs that are executed once and started immediately.
- **Delayed:** Jobs that are executed once but at a date and time in the future.
- **Recurring:** Jobs that are executed repeatedly at a regular CRON schedule.
- **Continuation:** Jobs that are executed on completion of a parent job.
- **Batches:** Jobs that are transactional. These are only available in the paid version.

Hangfire has persistent storage and can be configured to use:

- SQL Server
- Redis
- In-memory

- Community-developed storage

Let's set up an empty ASP.NET Core project to host Hangfire:

1. Use your preferred code editor to create a new Web API controller-based project, as defined in the following list:
 - Project template: **ASP.NET Core Empty** / web
 - Solution file and folder: `ModernServices`
 - Project file and folder:
`Northwind.Background.Hangfire`
 - **Configure for HTTPS**: Selected.
 - **Enable container support**: Cleared.
 - **Do not use top-level statements**: Cleared.
8. In the project file, treat warnings as errors, and add package references to work with Hangfire and persist its data to SQL Server, as shown in the following markup:

```
<ItemGroup> <PackageReference  
  Include="Hangfire.Core" />  
<PackageReference  
  Include="Hangfire.SqlServer" />  
<PackageReference  
  Include="Hangfire.AspNetCore" />  
<PackageReference  
  Include="Microsoft.Data.SqlClient" /> </  
ItemGroup>
```

9. Build the project to restore packages.
10. In the `Properties` folder, in `launchSettings.json`, modify the `applicationUrl` of the profile named `https` to use port `5115` for `https` and port `5116` for `http`, as highlighted in the following configuration:

```
"profiles": { ... "https" :{
  "commandName": "Project",
  "dotnetRunMessages": true,
  "launchBrowser": true, "applicationUrl"
  "https://localhost:5115;http://
  localhost:5116":, "environmentVariables":
  { "ASPNETCORE_ENVIRONMENT": "Development"
  }
}
```

11. In `appsettings.Development.json`, add an entry to set the Hangfire log level to `Information`, as highlighted in the following JSON:

```
{ "Logging": { "LogLevel": { "Default":
  "Information", "Microsoft.AspNetCore":
  "Warning", "Hangfire" "Information": } } }
```

12. Create a SQL Server database named `Northwind.HangfireDb`. If you are using SQL Server in a container, the server name will be `tcp:127.0.0.1,1433`.
 - If you are using Visual Studio, navigate to **View | Server Explorer**, right-click **Data Connections**, choose **Create New SQL Server database...**, enter connection information and the database name, and then click **OK**.
 - If you are using VS Code, navigate to **SQL Server**, right-click and choose **New Query**, enter connection information, and then in the query window, enter the following SQL command and execute it:

```
USE master GO CREATE DATABASE
[Northwind.HangfireDb] GO
```

15. In `Program.cs`, delete the existing statements, and then add statements to configure Hangfire to use SQL Server and to enable Hangfire Dashboard, as shown in the following code:

```
using Microsoft.Data.SqlClient; // To use
SqlConnectionStringBuilder. using
Hangfire; // To use GlobalConfiguration.
SqlConnectionStringBuilder connection =
new(); connection.InitialCatalog =
"Northwind.HangfireDb";
connection.MultipleActiveResultSets =
true; connection.Encrypt = true;
connection.TrustServerCertificate = true;
connection.ConnectTimeout = 5; // Default
is 30 seconds. connection.DataSource =
"tcp:127.0.0.1,1443"; // To use SQL Server
in a container. // To use SQL Server
authentication. connection.UserID =
Environment
.GetEnvironmentVariable("MY_SQL_USR");
connection.Password = Environment
.GetEnvironmentVariable("MY_SQL_PWD");
connection.PersistSecurityInfo = false;
var builder =
WebApplication.CreateBuilder(args);
builder.Services.AddHangfire(config =>
config
.SetDataCompatibilityLevel(CompatibilityLevel.Ver
.UseSimpleAssemblyNameTypeSerializer()
.UseRecommendedSerializerSettings()
.UseSqlServerStorage(connection.ConnectionString)
builder.Services.AddHangfireServer(); var
app = builder.Build();
app.UseHangfireDashboard();
app.MapGet("/", () => "Navigate to /
```



```
hangfire to see the Hangfire Dashboard.");  
app.MapHangfireDashboard(); app.Run();
```

16. Start the `Northwind.Background.Hangfire` project without debugging, and note the messages written to the console, as shown in the following output:

```
info:  
Hangfire.SqlServer.SqlServerObjectsInstaller[0]  
Start installing Hangfire SQL objects...  
info:  
Hangfire.SqlServer.SqlServerObjectsInstaller[0]  
Hangfire SQL objects installed. info:  
Microsoft.Hosting.Lifetime[14] Now  
listening on: https://localhost:5115 info:  
Microsoft.Hosting.Lifetime[14] Now  
listening on: http://localhost:5116 info:  
Hangfire.BackgroundJobServer[0] Starting  
Hangfire Server using job storage: 'SQL  
Server: .@Northwind.HangfireDb' info:  
Hangfire.BackgroundJobServer[0] Using the  
following options for SQL Server job  
storage: Queue poll interval: 00:00:00.  
info: Hangfire.BackgroundJobServer[0]  
Using the following options for Hangfire  
Server: Worker count: 20 Listening queues:  
'default' Shutdown timeout: 00:00:15  
Schedule polling interval: 00:00:15 info:  
Microsoft.Hosting.Lifetime[0] Application  
started. Press Ctrl+C to shut down. info:  
Microsoft.Hosting.Lifetime[0] Hosting  
environment: Development info:  
Microsoft.Hosting.Lifetime[0] Content root  
path: C:\apps-services-
```

```
net10\ModernServices
\Northwind.Background.Hangfire info:
Hangfire.Server.BackgroundServerProcess[0]
Server desktop-jlpqhr7:14120:c8ea792b
successfully announced in 140.4628 ms
info:
Hangfire.Server.BackgroundServerProcess[0]
Server desktop-jlpqhr7:14120:c8ea792b is
starting the registered dispatchers:
ServerWatchdog,
ServerJobCancellationWatcher,
ExpirationManager, CountersAggregator,
SqlServerHeartbeatProcess, Worker,
DelayedJobScheduler,
RecurringJobScheduler... info:
Hangfire.Server.BackgroundServerProcess[0]
Server desktop-jlpqhr7:14120:c8ea792b all
the dispatchers started
```

17. In the browser, note the plain text message, and then in the address bar, append `/hangfire`, and note the **Hangfire Dashboard** user interface, as shown in *Figure 11.7*:

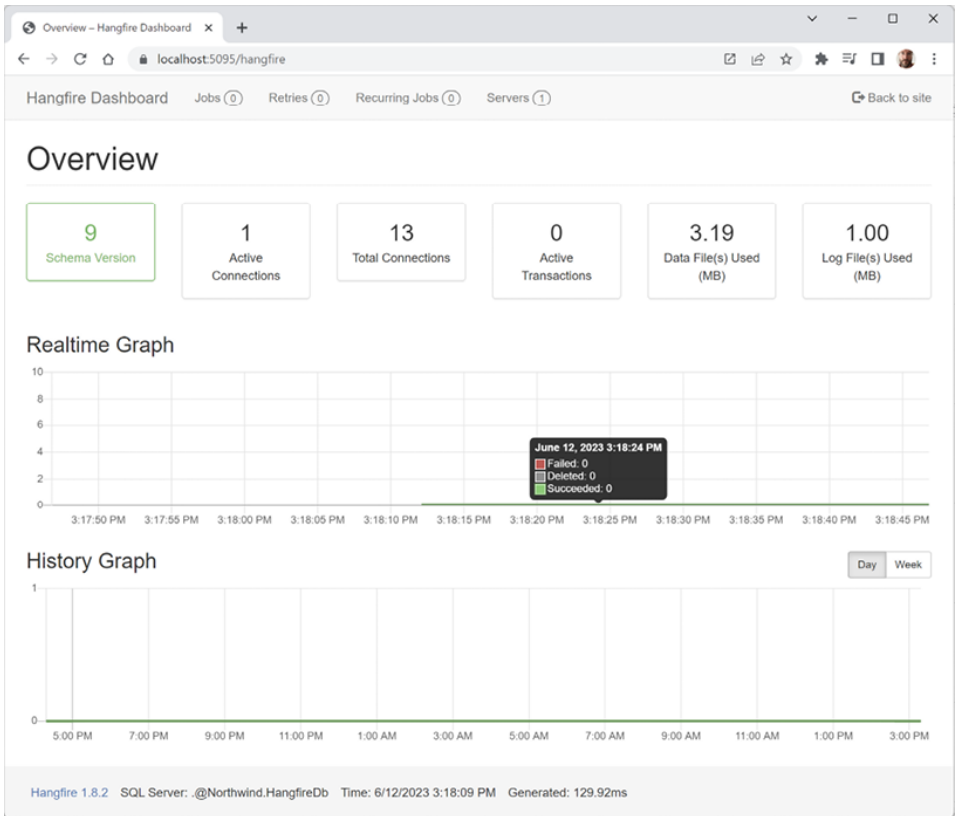


Figure 11.7: Hangfire Dashboard user interface

1. Close the browser window, and at the command prompt or terminal for the Hangfire service, press `Ctrl + C` to cleanly shut down the server, and note the messages, as shown in the following output:

```
info: Microsoft.Hosting.Lifetime[0]
Application is shutting down... info:
Hangfire.Server.BackgroundServerProcess[0]
Server desktop-j1pqhr7:14120:c8ea792b
caught stopping signal... info:
Hangfire.Server.BackgroundServerProcess[0]
Server desktop-j1pqhr7:14120:c8ea792b All
dispatchers stopped info:
```

```
Hangfire.Server.BackgroundServerProcess[0]  
Server desktop-j1pqhr7:14120:c8ea792b  
successfully reported itself as stopped in  
2.8874 ms info:  
Hangfire.Server.BackgroundServerProcess[0]  
Server desktop-j1pqhr7:14120:c8ea792b has  
been stopped in total 19.6204 ms
```

Scheduling jobs using Hangfire

Next, we will allow a client to schedule a job (in this case, just writing a message to the console a specified number of seconds in the future) by POSTing to the web service:

1. In the `Northwind.Background.Hangfire` project, add a new class file named `WriteMessageJobDetail.cs`.
2. In `WriteMessageJobDetail.cs`, define a class to represent a scheduled job, as shown in the following code:

```
namespace Northwind.Background.Models;  
public class WriteMessageJobDetail {  
    public string? Message { get; set; }  
    public int Seconds { get; set; } }
```

3. In the `Northwind.Background.Hangfire` project, add a new class file named `Program.Methods.cs`.
4. In `Program.Methods.cs`, extend the partial `Program` class with a method to write a message to the console in green, as shown in the following code:

```
using static System.Console; partial class  
Program { public static void
```

```
WriteMessage(string? message) {
    ConsoleColor previousColor =
    ConsoleColor; ConsoleColor =
    ConsoleColor.Green; WriteLine(message);
    ConsoleColor = previousColor; } }
```

5. In `Program.cs`, import namespaces to work with the job, as shown in the following code:

```
using Northwind.Background.Models; // To
use WriteMessageJobDetail. using
Microsoft.AspNetCore.Mvc; // To use
[FromBody].
```

6. In `Program.cs`, after the statement to map a GET request, map a POST request to the relative path `/schedulejob`, get the job details from the body of the POST request, and use it to schedule a background job, using Hangfire to run at the specified time in seconds in the future, as shown in the following code:

```
app.MapPost("/schedulejob", ([FromBody]
WriteMessageJobDetail job) => {
    BackgroundJob.Schedule( methodCall: () =>
    WriteMessage(job.Message), enqueueAt:
    DateTimeOffset.UtcNow +
    TimeSpan.FromSeconds(job.Seconds)); });
```

7. Start the `Northwind.Background.Hangfire` project without debugging.
8. In the command prompt or terminal, confirm that all the dispatchers started, as shown in the following output:

```
info:
Hangfire.Server.BackgroundServerProcess[0]
Server desktop-j1pqhr7:13916:9f1851b5 all
the dispatchers started
```

9. In the browser, navigate to `/hangfire` to view Hangfire Dashboard.
10. In your code editor, in the `HttpRequests` folder, create a file named `hangfire-schedule-job.http`.
11. In `hangfire-schedule-job.http`, add statements to make a POST request to the Hangfire service, as shown in the following code:

```
### Configure a variable for the Hangfire
web service base address. @base_address =
https://localhost:5115/ POST
{{base_address}}schedulejob Content-Type:
application/json { "message": "Hangfire is
awesome!", "seconds": 30 }
```

12. Send the request, and note the successful response, as shown in the following output:

```
HTTP/1.1 200 OK Content-Length: 0
Connection: close Date: Mon, 12 Jun 2023
16:04:20 GMT Server: Kestrel Alt-Svc:
h3=":5115"; ma=86400
```

13. In the browser, in **Hangfire Dashboard**, click **Jobs** in the top menu click **Jobs**, in the left side menu, and then note there is one scheduled job, as shown in *Figure 11.8*:

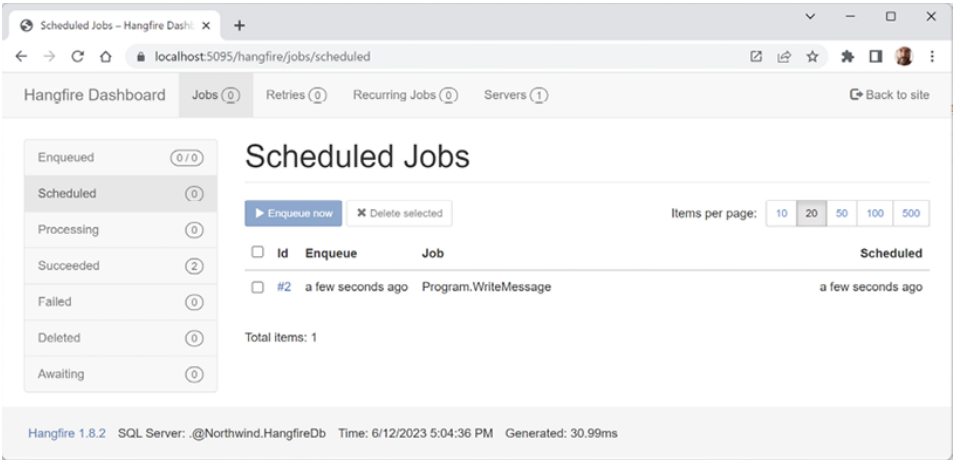


Figure 11.8: Scheduled jobs in Hangfire

1. Wait until 30 seconds have passed, and then in the left -side menu, click **Succeeded**, and note the job has succeeded, as shown in Figure 11.9:

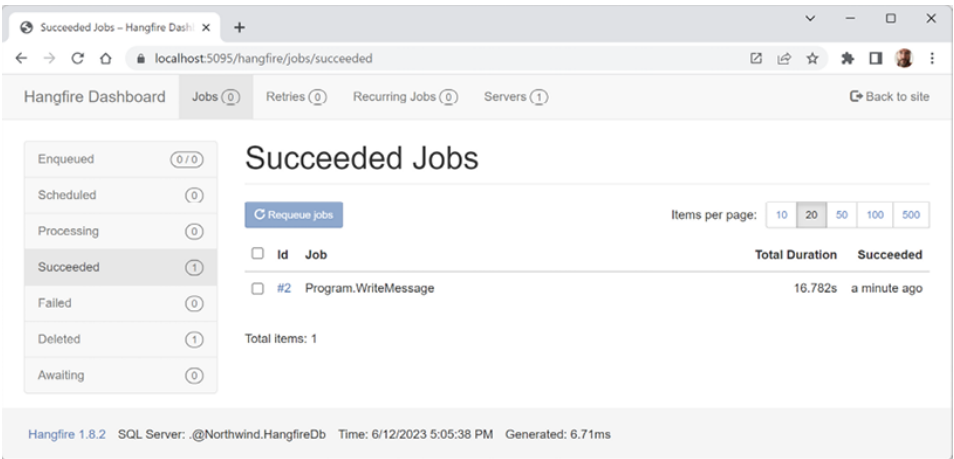


Figure 11.9: Succeeded jobs in Hangfire

1. At the command prompt or terminal, note the message that was written to the console, as shown in the following output:

```
Hangfire is awesome!
```

2. Close the browser and shut down the server.

You can read more about Hangfire at the following link: <https://www.hangfire.io/>.

Practicing and exploring

Test your knowledge and understanding by answering some questions, getting some hands-on practice, and exploring this chapter's topics with deeper research.

Exercise 11.1 – Online material

Online material can be extra content written by me for this book, or it can be references to content created by Microsoft or third parties.

Review the Reliable Web App pattern

The **Reliable Web App (RWA)** pattern is a set of best practices with prescriptive guidance that helps developers successfully migrate an on-premises web project to the cloud. It includes a reference implementation and shows how to make the most of Azure cloud services to modernize mission-critical workloads in a reliable, secure, high-performance, cost-efficient manner using modern design, development, and operational practices: <https://learn.microsoft.com/en-us/azure/architecture/web-apps/guides/reliable-web-app/dotnet/plan-implementation>.

A collection of videos about the RWA pattern for .NET is at the following link: <https://www.youtube.com/playlist?list=PLI7iePan8aH54gIDJquV61dE3ENyaDi3Q>.

Exercise 11.2 – Practice exercises

Practice exercises are the best way to reinforce skills and knowledge.

Replace the Distributed Memory Cache with another distributed cache implementation

In this chapter, we used the Distributed Memory Cache implementation to explore how to use a distributed cache.

As an optional exercise, in Docker, create a Redis container, and change your web service project configuration to use it.

In the `Northwind.WebApi.Service`, you will need to reference the Redis package, comment out the previously registered distributed cache implementation, and then call the extension method to register Redis as the distributed cache implementation:

```
//  
builder.Services.AddDistributedMemoryCache();  
builder.Services.AddStackExchangeRedisCache(options  
=> { options.Configuration =  
builder.Configuration  
.GetConnectionString("MyRedisConStr");  
options.InstanceName = "SampleInstance"; });
```

Read more at the following links:

- Azure Cache for Redis: <https://azure.microsoft.com/en-us/products/cache/>
- Redis NuGet package: <https://www.nuget.org/packages/Microsoft.Extensions.Caching.StackExchangeRedis>
- Redis with .NET: https://docs.redis.com/latest/rs/references/client_references/client_csharp

Replace Hangfire with Quartz.NET

Quartz.NET is a similar library to Hangfire. Read the official documentation, and then create a project named `Northwind.Background.Quartz` that implements the same functionality as

`Northwind.Background.Hangfire`: <https://www.quartz-scheduler.net/>.

Exercise 11.3 – Test your knowledge

Answer the following questions:

1. How much longer does it take to read 1 MB of data from an SSD compared to memory?
2. What is the difference between absolute and sliding expirations?
3. What unit of measurement is used by `Size` for the in-memory cache?
4. You have written the following statement to get information about in-memory caching but `stats` is `null`. What must you do to fix this issue?

```
MemoryCacheStatistics? stats =  
    _memoryCache.GetCurrentStatistics();
```

5. What data types can be stored in (a) an in-memory cache, and (b) a distributed cache?
6. What are the differences between the Retry and Circuit Breaker patterns?
7. When using the RabbitMQ default direct exchange, what must the routing key be for a queue named `product`?
8. What is the difference between a fanout and a topic exchange?
9. What port does RabbitMQ listen on by default?
10. When inheriting from the `BackgroundService` class, what method must you override that is called automatically by the host to run your service?

Appendix A, Answers to the Test Your Knowledge Questions, is available to download from the following link: https://github.com/markjprice/apps-services-net10/blob/main/docs/B31467_Appendix%20A.pdf.

Exercise 11.4 – Explore topics

Use the links on the following page to learn more details about the topics covered in this chapter: <https://github.com/markjprice/apps-services-net10/blob/main/docs/book-links.md#chapter-11---caching-queuing-and-resilient-background-services>.

Summary

In this chapter, you learned:

- About service architecture and how different parts of a system can affect performance.
- How to cache data closer to the action, using in-memory and distributed caching.
- How to control HTTP caching for clients and intermediaries.
- How to implement fault tolerance using Polly.
- How to implement queuing using RabbitMQ.
- How to implement long-running background services using `BackgroundService` and Hangfire.

In the next chapter, you will learn how to use SignalR to implement live communication between services and apps.

Get This Book **UNLOCK NOW** **ion and**

Exclusive

Scan the QR code
book by name, co



Search for this
ps on the page.

Note: *Keep your invoice handy. Purchases made directly from Packt don't require one.*

12

Broadcasting Real-Time Communication Using SignalR

In this chapter, you will be introduced to SignalR, a technology that enables a developer to create a service that can have multiple clients and broadcast messages to all of them or a subset of them live in real time. The canonical example is a group chat app. Other examples include notification systems and dashboards that need instantly up-to-date information like stock prices.

This chapter will cover the following topics:

- Understanding SignalR
- Building a live communication service using SignalR
- Building a web client using the SignalR JavaScript library
- Building a .NET console app SignalR client
- Streaming data using SignalR

Understanding SignalR

To understand the problem that SignalR solves, we need to understand what web development is like without it. The foundation of the web is HTTP, which for more than 30 years has been great for building general-purpose websites and

services. However, the web was not designed for specialized scenarios that require a web page to be instantaneously updated with new information as it becomes available. It helps to know the history of HTTP and how organizations worked to make it better for real-time communication between clients and servers.

The history of real-time communication on the web

In the early days of the web in the 1990s, browsers had to make a full-page HTTP GET request to the web server to get fresh information to show to the visitor.

In late 1999, Microsoft released Internet Explorer 5 with a component named **XMLHttpRequest** that could make asynchronous HTTP calls in the background. This, alongside **dynamic HTML (DHTML)**, allowed parts of the web page to be updated with fresh data smoothly.

The benefits of this technique were obvious, and soon, all browsers added the same component.

Google took maximum advantage of this capability to build clever web applications such as Google Maps and Gmail. A few years later, the technique became popularly known as **Asynchronous JavaScript and XML (AJAX)**.

AJAX still uses HTTP to communicate, however, and that has limitations:

- First, HTTP is a request-response communication protocol, meaning that the server cannot push data to the client. It must wait for the client to make a request.
- Second, HTTP request and response messages have headers with lots of potentially unnecessary overhead.

WebSocket is full-duplex, meaning that either the client or server can initiate the communication of new data. WebSocket uses the same TCP connection for the life cycle of the connection. It is also more efficient in the message sizes that

it sends because they are minimally framed with 2 bytes.

WebSocket works over HTTP ports 80 and 443 so it is compatible with the HTTP protocol, and the WebSocket handshake uses the HTTP **Upgrade** header to switch from the HTTP protocol to the WebSocket protocol.

Modern web apps are expected to deliver up-to-date information. Live chat is the canonical example, but there are lots of potential applications, from stock prices to games.

Whenever you need the server to push updates to the web page, you need a web-compatible, real-time communication technology. WebSocket could be used, but it is not supported by all clients. You can check which clients support WebSocket using the web page found at the following link: <https://caniuse.com/websockets>.

WebSocket or WebSockets? “The **WebSocket** protocol was standardized by the IETF as RFC 6455 in 2011. The current API specification allowing web applications to use this protocol is known as *WebSockets*.” From the Wikipedia page found at the following link: <https://en.wikipedia.org/wiki/WebSocket>.

Introducing SignalR

ASP.NET Core SignalR is an open-source library that simplifies adding real-time web functionality to apps by providing an abstraction over multiple underlying communication technologies, which allows you to add real-time communication capabilities using C# code.

The developer does not need to understand or implement the underlying technology used, and SignalR will automatically switch between underlying technologies depending on what the visitor’s web browser supports. For example, SignalR will use WebSocket when it’s available and gracefully fall

back on other technologies, such as AJAX long polling, when it isn't, while your application code stays the same.

SignalR is an API for server-to-client **remote procedure calls (RPCs)**. The RPCs call JavaScript functions on clients from server-side .NET code. SignalR has hubs to define the pipeline and handles the message dispatching automatically using two built-in hub protocols: JSON and a binary one based on MessagePack.

On the server side, SignalR runs everywhere that ASP.NET Core runs: Windows, macOS, or Linux servers. SignalR supports the following client platforms:

- JavaScript clients for current browsers including Chrome, Firefox, Safari, and Edge.
- .NET clients including Blazor, .NET MAUI, and Xamarin for Android and iOS mobile apps.
- Java 8 and later.

Azure SignalR Service

Earlier, I mentioned that it would be good practice to separate the SignalR service hosting project from the web project that uses the JavaScript library to act as a client. This is because a SignalR service potentially needs to handle lots of simultaneous client requests and respond quickly to them all.

Once you separate the SignalR hosting, you can take advantage of **Azure SignalR Service**. This offers global reach and a world-class data center and network. Plus, it scales to millions of connections while meeting SLAs, like providing compliance and high security.

You can learn more about Azure SignalR Service at the following link: <https://learn.microsoft.com/en-us/azure/azure-signalr/signalr-overview>.

Designing method signatures

When designing the method signatures for a SignalR service, it is good practice to define methods with a single message parameter rather than multiple simple type parameters. This good practice is not enforced by the technology with SignalR, so you will have to be disciplined.

For example, instead of passing multiple `string` (or other type) values, define a type with multiple properties to use as the single `Message` parameter, as shown in the following code:

```
// Bad practice: RPC method with multiple  
parameters. public void SendMessage(string to,  
string body) // Good practice: single  
parameter using a complex type. public class  
Message { public string To { get; set; }  
public string Body { get; set; } } public void  
SendMessage(Message message)
```

The reason for this good practice is that it allows future changes like adding a third property for the message `Title`. For the bad practice example, a third `string` parameter named `title` would need to be added and existing clients would get errors because they are not sending the extra `string` value. But using the good practice example will not break the method signature so existing clients can continue to call it as before the change. On the server side, the extra `Title` property will just have a `null` value that can be checked for, and perhaps be set to a default value.

SignalR method parameters are serialized as JSON, so all nested objects are accessible in JavaScript if needed.

Now that we've explored the fundamentals of SignalR and its various aspects, like good practices for method signature design, let's walk through how to build a live communication service using SignalR.

Building a live communication service using SignalR

The SignalR *server* library is included in ASP.NET Core, but the JavaScript *client* library is not automatically included in the project. Remember, SignalR supports multiple client types, and a web page using JavaScript is just one of them.

We will use the **Library Manager CLI** to get the client library from **unpkg**, a **content delivery network (CDN)** that can deliver anything found in the Node.js package manager.

Let's add a SignalR server-side hub and client-side JavaScript to an ASP.NET Core MVC project to implement a chat feature that allows visitors to send messages to:

- Everyone currently using the website.
- Dynamically defined groups.
- A single specified user.

Good practice: In a production solution, it would be better to host the SignalR hub in a separate web project so that it can be hosted and scaled independently from the rest of the website. Live communication can often put an excessive load on a website.

Defining some shared models

First, we will define two shared models that can be used on both the server-side and client-side .NET projects that will work with our chat service:

1. Use your preferred code editor to create a new project, as defined in the following list:
 - Project template: **Class Library** / `classlib`

- Solution file and folder: ModernServices
- Project file and folder: Northwind.Common

5. In the Northwind.Common project, rename the Class1.cs file to UserModel.cs.
6. Modify its contents to define a model for registering a user's name, unique connection ID, and the groups that they belong to, as shown in the following code:

```
namespace Northwind.Chat.Models; public
class UserModel { public string Name {
get; set; } = null!; public string
ConnectionId { get; set; } = null!; public
string? Groups { get; set; } // comma-
separated list }
```

Good practice: In a real-world app, you would want to use a collection of string values for the Groups property, but this coding task is not about how to provide a web user experience for editing multiple string values. We will provide a simple text box instead and focus on learning SignalR.

1. In the Northwind.Common project, add a class file named MessageModel.cs. Modify its contents to define a message model with properties for whom the message is sent to and who the message was sent from, and the message body, as shown in the following code:

```
namespace Northwind.Chat.Models; public
class MessageModel { public string From {
```

```
get; set; } = null!; public string To {  
get; set; } = null!; public string? Body {  
get; set; } }
```

Enabling a server-side SignalR hub

Next, we will enable a SignalR hub on the server side in an ASP.NET Core MVC project:

1. Use your preferred code editor to add a new project, as defined in the following list:
 - Project template: **ASP.NET Core Web App (Model-View-Controller)** / `mvc`
 - Solution file and folder: `ModernServices`
 - Project file and folder:
`Northwind.SignalR.Service`
 - **Authentication type:** `None`
 - **Configure for HTTPS:** `Selected`
 - **Enable container support:** `Cleared`
 - **Do not use top-level statements:** `Cleared`
9. In the `Northwind.SignalR.Service` project, treat warnings as errors and add a project reference to the `Northwind.Common` project, as shown in the following markup:

```
<ItemGroup> <ProjectReference Include= "..  
\Northwind.Common\Northwind.Common.csproj"  
> </ItemGroup>
```

10. In the `Properties` folder, in `launchSettings.json`, in the `https` profile, modify the `applicationUrl` to use port `5121` for `https` and `5122` for `http`, as shown highlighted in the following

configuration:

```
"https": { "commandName": "Project",
"dotnetRunMessages": true,
"launchBrowser": true, "applicationUrl"
"https://localhost:5121;http://
localhost:5122":, "environmentVariables":
{ "ASPNETCORE_ENVIRONMENT": "Development"
}
```

11. In the `Northwind.SignalR.Service` project, add a new folder named `Hubs`.
12. In the `Hubs` folder, add a class file named `ChatHub.cs`.
13. In `ChatHub.cs`, modify its contents to inherit from the `Hub` class and implement two methods that can be called by a client, as shown in the following code:

To save time entering code, the whole file is available at the following link:

<https://github.com/markjprice/apps-services-net10/blob/main/code/ModernServices/Northwind.SignalR.Service/Hubs/ChatHub.cs>.

```
using Microsoft.AspNetCore.SignalR; // To use
IClientProxy. using Northwind.Chat.Models; //
To use UserModel, MessageModel. namespace
Northwind.SignalR.Service.Hubs; public class
ChatHub : Microsoft.AspNetCore.SignalR.Hub {
// A new instance of ChatHub is created to
```

```
process each method so we // must store user
names, connection IDs, and groups in a static
field. private static Dictionary<string,
UserModel> Users = new(); public async Task
Register(UserModel newUser) { UserModel user;
string action = "registered as a new user"; //
Try to get a stored user with a match on new
user. if (Users.ContainsKey(newUser.Name)) {
user = Users[newUser.Name]; // Remove any
existing group registrations. if (user.Groups
is not null) { foreach (string group in
user.Groups.Split(',')) { await
Groups.RemoveFromGroupAsync(
user.ConnectionId, group); } } user.Groups =
newUser.Groups; // Connection ID might have
changed if the browser // refreshed so update
it. user.ConnectionId = Context.ConnectionId;
action = "updated your registered user"; }
else { if (string.IsNullOrEmpty(newUser.Name))
{ // Assign a GUID for name if they are
anonymous. newUser.Name =
Guid.NewGuid().ToString(); }
newUser.ConnectionId = Context.ConnectionId;
Users.Add(key: newUser.Name, value: newUser);
user = newUser; } if (user.Groups is not null)
{ // A user does not have to belong to any
groups // but if they do, register them with
the Hub. foreach (string group in
user.Groups.Split(',')) { await
Groups.AddToGroupAsync( user.ConnectionId,
group); } } // Send a message to the
registering user informing of success.
MessageModel message = new() { From = "SignalR
Hub", To = user.Name, Body = string.Format(
"You have successfully {0} with connection ID
```

```

{1}."", arg0: action, arg1: user.ConnectionId)
}; IClientProxy proxy =
Clients.Client(user.ConnectionId); await
proxy.SendAsync("ReceiveMessage", message); }
public async Task SendMessage(MessageModel
message) { IClientProxy proxy; if
(string.IsNullOrEmpty(message.To)) {
message.To = "Everyone"; proxy = Clients.All;
await proxy.SendAsync("ReceiveMessage",
message); return; } // Split To into a list of
user and group names. string[]
userAndGroupList = message.To.Split(','); //
Each item could be a user or group name.
foreach (string userOrGroup in
userAndGroupList) { if
(Users.ContainsKey(userOrGroup)) { // If the
item is in Users then send the message to that
user // by looking up their connection ID in
the dictionary. message.To = $"User:
{Users[userOrGroup].Name}"; proxy =
Clients.Client(Users[userOrGroup].ConnectionId);
} else // Assume the item is a group name to
send the message to. { message.To = $"Group:
{userOrGroup}"; proxy =
Clients.Group(userOrGroup); } await
proxy.SendAsync("ReceiveMessage", message); }
} }

```

Note the following:

- ChatHub has a private field to store a list of registered users. It is a dictionary with their names as unique keys.
- ChatHub has two methods that a client can call: Register and SendMessage.
- Register has a single parameter of type UserModel. The

user's name, connection ID, and groups are stored in the static dictionary so that the user's name can be used to look up the connection ID later and send messages directly to that one user. After registering a new user or updating the registration of an existing user, a message is sent back to the client informing them of success.

- `SendMessage` has a single parameter of type `MessageModel`. The method branches based on the value of the `To` property. If `To` does not have a value, it calls the `All` property to get a proxy that will communicate with every client. If `To` has a value, the `string` is split using comma separators into an array. Each item in the array is checked to see if it matches a user in `Users`. If it matches, it calls the `Client` method to get a proxy that will communicate just with that one client. If it does not match, the item might be a group, so it calls the `Group` method to get a proxy that will communicate with just the members of that group. Finally, it sends the message asynchronously using the proxy.

1. In `Program.cs`, import the namespace for your SignalR hub, as shown in the following code:

```
using Northwind.SignalR.Service.Hubs; //
To use ChatHub.
```

2. In the section that configures services, add a statement to add support for SignalR to the services collection, as shown in the following code:

```
builder.Services.AddSignalR();
```

3. In the section that configures the HTTP pipeline, before the call to `map`

controller routes, add a statement to map the relative URL path `/chat` to your SignalR hub, as shown in the following code:

```
app.MapHub<ChatHub>("/chat");
```

Now that we have built the host for a SignalR service hub we can build clients that use it. We will start with a web client that will use the SignalR JavaScript library.

Building a web client using the SignalR JavaScript library

We will add the SignalR client-side JavaScript library so that we can use it on a web page:

1. At a command prompt or terminal for the `Northwind.SignalR.Service` project/folder, install the Library Manager CLI as a global tool, as shown in the following command:

```
dotnet tool install -g  
Microsoft.Web.LibraryManager.Cli
```

This tool might already be installed globally. To update it to the latest version, repeat the command but replace `install` with `update`.

1. Enter a command to add the `signalr.js` and `signalr.min.js` libraries to the project from the `unpkg` source, as shown in the following command:

```
libman install @microsoft/signalr@latest -  
p unpkg -d wwwroot/js/signalr --files  
dist/browser/signalr.js --files dist/  
browser/signalr.min.js
```

Never copy long commands from a PDF and paste them directly to the command prompt. Always clean them up in a basic text editor to remove extraneous new lines and so on, and then recopy them. To make it easier to enter long command lines, you can copy them from the following link:

[https://github.com/
markjprice/apps-services-
net10/blob/main/docs/
command-lines.md](https://github.com/markjprice/apps-services-net10/blob/main/docs/command-lines.md)

1. Note the success message, as shown in the following output:

```
wwwroot/js/signalr/dist/browser/signalr.js  
written to disk wwwroot/js/signalr/dist/  
browser/signalr.min.js written to disk  
Installed library "@microsoft/  
signalr@latest" to "wwwroot/js/signalr"
```

Visual Studio also has a GUI for adding client-side JavaScript libraries. To use it, right-click a web project and then navigate to **Add | Client Side Libraries**.

Adding a chat page to the MVC website

Next, we will add chat functionality to the home page:

1. In Views/Home, in Index.cshtml, modify its contents, as shown in the following markup:

```
@using Northwind.Chat.Models @{
    ViewData["Title"] = "SignalR Chat"; } <div
class="container">
<h1>@ViewData["Title"]</h1> <hr /> <div
class="row"> <div class="col">
<h2>Register User</h2> <div class="mb-3">
<label for="myName" class="form-label">My
name</label> <input type="text"
class="form-control" id="myName"
value="Alice" required /> </div> <div
class="mb-3"> <label for="myGroups"
class="form-label">My groups</label>
<input type="text" class="form-control"
id="myGroups" value="Sales,IT" /> </div>
<div class="mb-3"> <input type="button"
class="form-control" id="registerButton"
value="Register User" /> </div> </div>
<div class="col"> <h2>Send Message</h2>
<div class="mb-3"> <label for="from"
class="form-label">From</label> <input
type="text" class="form-control" id="from"
value="Alice" readonly /> </div> <div
class="mb-3"> <label for="to" class="form-
label">To</label> <input type="text"
class="form-control" id="to" /> </div>
<div class="mb-3"> <label for="body"
class="form-label">Body</label> <input
type="text" class="form-control" id="body"
/> </div> <div class="mb-3"> <input
```

```

type="button" class="form-control"
id="sendButton" value="Send Message" /> </
div> </div> </div> <div class="row"> <div
class="col"> <hr /> <h2>Messages
received</h2> <ul id="messages"></ul> </
div> </div> </div> <script src="~/js/
signalr/dist/browser/signalr.js"></script>
<script src="~/js/chat.js"></script>

```

Note the following:

- There are three sections on the page: **Register User**, **Send Message**, and **Messages received**.
- The **Register User** section has two inputs for the visitor's name, a comma-separated list of the groups that they want to be a member of, and a button to click to register.
- The **Send Message** section has three inputs for the name of the user that the message is from, the names of users and groups that the message will be sent to, and the body of the message. Plus, there is a button to click to send the message.
- The **Messages received** section has a bullet list element that will be dynamically populated with a list item when a message is received.
- There are two script elements for the SignalR JavaScript client-side library and the JavaScript implementation of the chat client.

1. In `wwwroot/js`, add a new JavaScript file named `chat.js` and modify its contents, as shown in the following code:

```

"use strict"; var connection = new
signalR.HubConnectionBuilder() .withUrl("/
chat") .build();
document.getElementById("registerButton").disable

```

```

= true;
document.getElementById("sendButton").disabled
= true;
document.getElementById("myName").addEventListener
function () {
document.getElementById("from").value =
document.getElementById("myName").value; }
); connection.start().then(function () {
document.getElementById("registerButton").disabled
= false;
document.getElementById("sendButton").disabled
= false; }).catch(function (err) { return
console.error(err.toString()); });
connection.on("ReceiveMessage", function
(received) { var li =
document.createElement("li");
document.getElementById("messages").appendChild(li);
li.textContent = // This string must use
backticks ` to enable an interpolated //
string. If you use single quotes ' then it
will not work! `To ${received.to}, From
${received.from}: ${received.body}`; });
document.getElementById("registerButton").addEventListener
function (event) { var registermodel = {
name:
document.getElementById("myName").value,
groups:
document.getElementById("myGroups").value
}; connection.invoke("Register",
registermodel).catch(function (err) {
return console.error(err.toString()); });
event.preventDefault(); });
document.getElementById("sendButton").addEventListener
function (event) { var messagemodel = {
from:

```

```
document.getElementById("from").value, to:
document.getElementById("to").value, body:
document.getElementById("body").value };
connection.invoke("SendMessage",
messagemodel).catch(function (err) {
return console.error(err.toString()); });
event.preventDefault(); });
```

Note the following about the preceding code:

- The script creates a SignalR hub connection builder specifying the relative URL path to the chat hub on the server /chat.
- The script disables the **Register** and **Send** buttons until the connection is successfully established with the server-side hub.
- An `input` event handler is added to the **My name** text box to keep it synchronized with the **From** text box.
- When the connection gets a `ReceiveMessage` call from the server-side hub, it adds a list item element to the `messages` bullet list. The content of the list item contains details of the message like `from`, `to`, and `body`. For the two models that we defined in C#, note that JavaScript uses camelCase compared to C#, which uses PascalCase.

The message is formatted using a JavaScript interpolated string. This feature requires backticks ``` at the start and end of the string value and the use of curly brackets `${}` for dynamic placeholders.

- A `click` event handler is added to the **Register User** button that creates a register model with the user's name and their groups and then invokes the `Register` method on the server side.

- A `click` event handler is added to the **Send Message** button that creates a message model with the `from`, `to`, and `body`, and then invokes the `SendMessage` method on the server side.

Testing the chat feature

Now we are ready to try sending chat messages between multiple website visitors:

1. Start the `Northwind.SignalR.Service` project website using the `https` profile:

- If you are using Visual Studio, then select the **https** profile in the toolbar, and then start the `Northwind.SignalR.Service` project without debugging.
- If you are using VS Code, then at the command prompt or terminal, enter the following command:

```
dotnet run --launch-profile  
https
```

- On Windows, if Windows Defender Firewall blocks access, then click **Allow access**.
5. Start Chrome and navigate to <https://localhost:5121/>.
 6. Note that `Alice` is already entered for the name, and `Sales`, `IT` is already entered for her groups. Click **Register User**, and note the response back from the **SignalR Chat**, as shown in *Figure 12.1*:

SignalR Chat

Register User

My name

My groups

Send Message

From

To

Body

Messages received

- To Alice, From SignalR Hub: You have successfully registered as a new user with connection id L4lrQ67hM36lp0b9192kRw.

Figure 12.1: Registering a new user in chat

1. Open a new Chrome window or start another browser like Firefox or Edge.
2. Navigate to `https://localhost:5121/`.
3. Enter `Bob` for the name, `Sales` for his groups, and then click **Register User**.
4. Open a new Chrome window or start another browser like Firefox or Edge.
5. Navigate to `https://localhost:5121/`.
6. Enter `Charlie` for the name, `IT` for his groups, and then click **Register User**.
7. Arrange the browser windows so that you can see all three simultaneously.

A great tool for arranging windows is PowerToys and its FancyZones feature. Learn more at the following link:
<https://learn.microsoft.com/>

1. In Alice's browser, enter the following:

- **To:** Sales
- **Body:** Sell more!

4. Click **Send Message**.

5. Note that Alice and Bob receive the message, as shown in *Figure 12.2*:

The figure consists of three side-by-side screenshots of a web application titled "SignalR Chat".

- Left Screenshot:** Shows the "Register User" form with "My name" set to "Alice" and "My groups" set to "Sales, IT". Below it is a "Register User" button. To the right, the "Send Message" form has "From" set to "Alice", "To" set to "Sales", and "Body" set to "Sell more!". A "Send Message" button is at the bottom. The "Messages received" section shows two messages: "To Alice, From SignalR Hub: You have successfully registered as a new user with connection id QnOZIC-WojrrfmrN1MDqQ." and "To Group: Sales, From Alice: Sell more!".
- Middle Screenshot:** Shows the "Register User" form with "My name" set to "Bob" and "My groups" set to "Sales". Below it is a "Register User" button. The "Send Message" form is empty. The "Messages received" section shows two messages: "To Bob, From SignalR Hub: You have successfully registered as a new user with connection id f5oM07_GfP8JNZP47W8t6w." and "To Group: Sales, From Alice: Sell more!".
- Right Screenshot:** Shows the "Register User" form with "My name" set to "Charlie" and "My groups" set to "IT". Below it is a "Register User" button. The "Send Message" form is empty. The "Messages received" section shows one message: "To Charlie, From SignalR Hub: You have successfully registered as a new user with connection id 1FbXWC78lq4EYmN9g4kuQ."

Figure 12.2: Alice sends a message to the Sales group

1. In Bob's browser, enter the following:

- **To:** IT
- **Body:** Fix more bugs!

4. Click **Send Message**.

5. Note that Alice and Charlie receive the message, as shown in *Figure 12.3*:

SignalR Chat

Register User

My name

My groups

Register User

Send Message

From

To

Body

Send Message

Messages received

- To Alice, From SignalR Hub: You have successfully registered as a new user with connection id QnOZiC-WojrrfmrN1MDqQ.
- To Group: Sales, From Alice: Sell more!
- To Group: IT, From Bob: Fix more bugs!

SignalR Chat

Register User

My name

My groups

Register User

Send Message

From

To

Body

Send Message

Messages received

- To Bob, From SignalR Hub: You have successfully registered as a new user with connection id f5oM07_GiPBjNZP47W8t6w.
- To Group: Sales, From Alice: Sell more!

SignalR Chat

Register User

My name

My groups

Register User

Send Message

From

To

Body

Send Message

Messages received

- To Charlie, From SignalR Hub: You have successfully registered as a new user with connection id 1FbXlWC78lq4EYmN9g4kuQ.
- To Group: IT, From Bob: Fix more bugs!

Figure 12.3: Bob sends a message to the IT group

- In Alice's browser, enter the following:
 - To:** Bob
 - Body:** Bonjour Bob!
- Click **Send Message**.
- Note that only Bob receives the message.
- In Charlie's browser, enter the following:
 - To:** Leave it empty.
 - Body:** Everybody dance now!
- Click **Send Message**.
- Note that everyone receives the message, as shown in Figure 12.4:

SignalR Chat

Register User

My name

My groups

Register User

Send Message

From

To

Body

Send Message

Messages received

- To Alice, From SignalR Hub: You have successfully registered as a new user with connection id QnOZiC-WojrrfmrN1MDqQ.
- To Group: Sales, From Alice: Sell more!
- To Group: IT, From Bob: Fix more bugs!
- To Everyone, From Charlie: Everybody dance now!

SignalR Chat

Register User

My name

My groups

Register User

Send Message

From

To

Body

Send Message

Messages received

- To Bob, From SignalR Hub: You have successfully registered as a new user with connection id f5oM07_GiPBjNZP47W8t6w.
- To Group: Sales, From Alice: Sell more!
- To User: Bob, From Alice: Bonjour Bob!
- To Everyone, From Charlie: Everybody dance now!

SignalR Chat

Register User

My name

My groups

Register User

Send Message

From

To

Body

Send Message

Messages received

- To Charlie, From SignalR Hub: You have successfully registered as a new user with connection id 1FbXlWC78lq4EYmN9g4kuQ.
- To Group: IT, From Bob: Fix more bugs!
- To Everyone, From Charlie: Everybody dance now!

Figure 12.4: Charlie sends a message to everyone

1. In Charlie's browser, enter the following:
 - **To:** HR, Alice
 - **Body:** Is anyone in HR listening?
4. Click **Send Message**.
5. Note that Alice receives the message sent directly to her, but since the HR group does not exist, no one receives the message sent to that group, as shown in *Figure 12.5*:

The figure consists of three side-by-side screenshots of the 'SignalR Chat' web application, illustrating the state of the system after Charlie sends a message to 'HR, Alice'.

Left Screenshot (Alice's view): Alice is the user. The 'To' field contains 'Bob'. The 'Body' field contains 'Bonjour Bob!'. The 'Send Message' button is visible. The 'Messages received' section shows:

- To Alice, From SignalR Hub: You have successfully registered as a new user with connection id QnOZIC-WojrfrmrN1MDQ.
- To Group: Sales, From Alice: sell more!
- To Group: IT, From Bob: Fix more bugs!
- To Everyone, From Charlie: Everybody dance now!
- To User: Alice, From Charlie: Is anyone in HR listening?

Middle Screenshot (Bob's view): Bob is the user. The 'To' field contains 'IT'. The 'Body' field contains 'Fix more bugs!'. The 'Send Message' button is visible. The 'Messages received' section shows:

- To Bob, From SignalR Hub: You have successfully registered as a new user with connection id 5oM07_Gp8JNZ47W86w.
- To Group: Sales, From Alice: sell more!
- To User: Bob, From Alice: Bonjour Bob!
- To Everyone, From Charlie: Everybody dance now!

Right Screenshot (Charlie's view): Charlie is the user. The 'To' field contains 'HR, Alice'. The 'Body' field contains 'Is anyone in HR listening?'. The 'Send Message' button is visible. The 'Messages received' section shows:

- To Charlie, From SignalR Hub: You have successfully registered as a new user with connection id 1FbXWCTBiq4EYmN9g4kuQ.
- To Group: IT, From Bob: Fix more bugs!
- To Everyone, From Charlie: Everybody dance now!

Figure 12.5: Charlie sends a message to Alice and a group that does not exist

1. Close the browsers and shut down the web server.

You have now built a SignalR hub service and a web page that uses JavaScript to send messages between multiple clients. Next, let's see how any .NET project can participate as a SignalR client by creating a console app.

Building a .NET console app SignalR client

You have just seen a .NET service hosting a SignalR hub, and a JavaScript client exchanging messages with other clients via that SignalR hub. Now, let's

create a .NET client for SignalR.

Creating a .NET client for SignalR

We will use a console app, although any .NET project type would need the same package reference and implementation code:

1. Use your preferred code editor to add a new project, as defined in the following list:

- Project template: **Console Application** / console
- Solution file and folder: ModernServices
- Project file and folder:

Northwind.SignalR.Client.Console

5. Add a package reference for the ASP.NET Core SignalR client and a project reference for Northwind.Common, treat warnings as errors, and globally and statically import the System.Console class, as shown in the following markup:

```
<ItemGroup> <PackageReference  
Include="Microsoft.AspNetCore.SignalR.Client"  
> </ItemGroup> <ItemGroup>  
<ProjectReference Include= "..  
\Northwind.Common\Northwind.Common.csproj"  
> </ItemGroup>
```

6. Build the project to restore packages and build referenced projects.
7. In Program.cs, delete the existing statements, import namespaces for working with SignalR as a client and the chat models, and then add statements to prompt the user to enter a username and groups to register with, create a hub connection, and listen for received messages, as shown in the following code:

```

using Microsoft.AspNetCore.SignalR.Client;
// To use HubConnection. using
Northwind.Chat.Models; // To use
UserModel, MessageModel. Write("Enter a
username (required): "); string? username
= ReadLine(); if
(string.IsNullOrEmpty(username)) {
WriteLine("You must enter a username to
register with chat!"); return; }
Write("Enter your groups (optional): ");
string? groups = ReadLine(); HubConnection
hubConnection = new HubConnectionBuilder()
.WithUrl("https://localhost:5121/chat")
.Build();
hubConnection.On<MessageModel>("ReceiveMessage",
message => { WriteLine($"To {message.To},
From {message.From}: {message.Body}"); });
await hubConnection.StartAsync();
WriteLine("Successfully started.");
UserModel registration = new() { Name =
username, Groups = groups }; await
hubConnection.InvokeAsync("Register",
registration); WriteLine("Successfully
registered."); WriteLine("Listening...
(press ENTER to stop.)"); ReadLine();

```

Note the following about the preceding code:

- **Import namespaces.**

Microsoft.AspNetCore.SignalR.Client gives you the SignalR client API, specifically HubConnection and HubConnectionBuilder. Northwind.Chat.Models is your own shared project or NuGet package that contains: UserModel and MessageModel. This shared-model

approach is common and sensible. It avoids stringly-typed JSON nonsense and keeps client and server aligned.

- **The console prompts for a username and reads it from standard input.** If the user hits *ENTER* without typing anything, the app immediately exits.
- **Optional group membership.** This allows the user to specify one or more chat groups. How those groups are formatted and interpreted depends entirely on the server-side hub logic. Allowed patterns include comma-separated group names, a single group name, or leaving the group name empty, meaning no group.
- **Creating the SignalR connection.** The `HubConnectionBuilder` creates a SignalR client connection configured to talk to a hub endpoint at: <https://localhost:5121/chat>. `/chat` is the hub endpoint path configured on the server. This will negotiate transport automatically: WebSockets first, then fall back to Server-Sent Events or long polling if needed. HTTPS is required by default for modern SignalR setups.
- **Registering a handler for incoming messages.** `hubConnection.On<MessageModel>` subscribes to a server-to-client method call named `"ReceiveMessage"`, expects the server to send a payload that can be deserialized into `MessageModel`, and executes the lambda every time the server invokes that method.

Warning! If this handler is not registered before the connection starts, messages sent early could be missed.

- **Starting the connection.** This actually opens the connection to

the server. Under the hood, SignalR does HTTP negotiation, transport selection, connection establishment, and the protocol handshake. If this fails, an exception is thrown, and the app crashes unless you catch it. Production code should absolutely wrap this in `try/catch` and log failures.

- **Creating the registration payload.** `UserModel registration = new()` creates a user registration object that will be sent to the server. This assumes that the server hub has a method named `Register` and that method takes a `UserModel` as its parameter. This is classic RPC-style SignalR usage.
- **Calling a server method.**
`hubConnection.InvokeAsync("Register", registration)` invokes the server-side hub method called `Register`. Important distinction: `InvokeAsync` waits for the server method to complete, whereas `SendAsync` would fire-and-forget. Here we are explicitly waiting for the server to finish registering the user before continuing. On the server, the hub method does things like track connected users, add the connection to SignalR groups, and broadcast join notifications.
- **Waiting for messages.** The `ReadLine()` is a blocking read that keeps the process alive. While the app is blocked on `ReadLine()`, SignalR continues running on background threads, receiving messages and invoking `ReceiveMessage`. Pressing *ENTER* causes the program to exit, which implicitly tears down the connection.

In short, it is a headless SignalR chat client. No UI. No server logic. No persistence. Just a thin client that speaks SignalR and writes text to `stdout`. Now let's test it.

Testing the .NET console app client

Let's start the SignalR service and call it from the console app:

1. Start the `Northwind.SignalR.Service` project website using the `https` profile without debugging.
2. Start Chrome and navigate to `https://localhost:5121/`.
3. Click **Register User**.
4. Start the `Northwind.SignalR.Client.Console` project.
5. Enter your name and the groups: `Sales, Admins`.
6. Arrange the browser and console app windows so that you can see both simultaneously.
7. In Alice's browser, enter the following:
 - **To:** `Sales`
 - **Body:** `Go team!`
10. Click **Send Message**, and note that Alice and you receive the message, as shown in *Figure 12.6*:

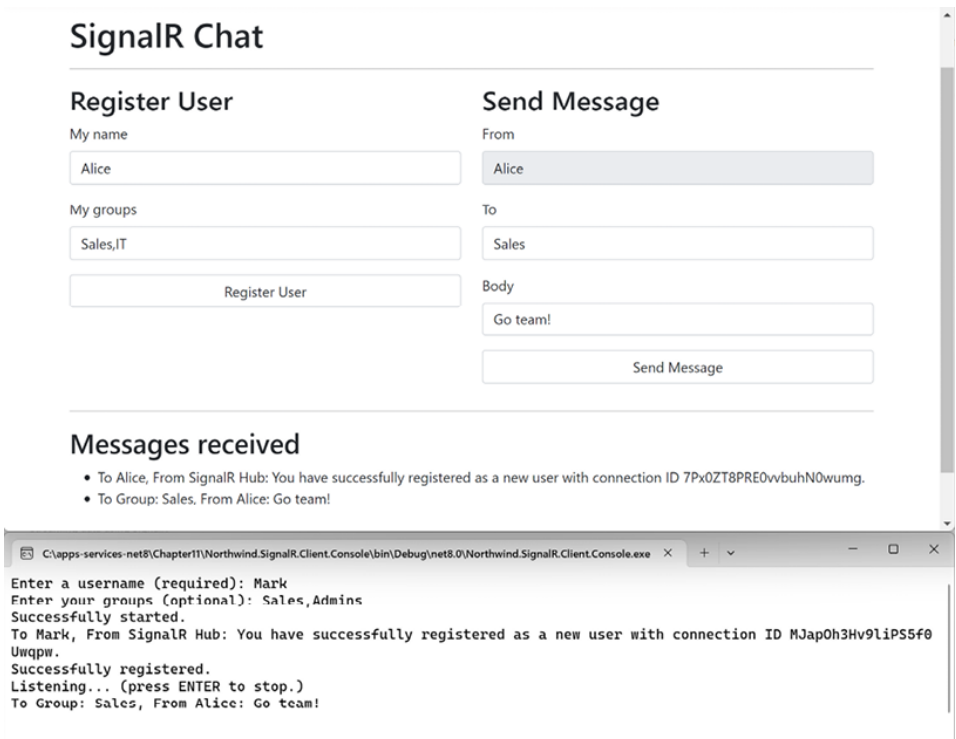


Figure 12.6: Alice sends a message to the Sales team including a user in a console app

1. In the console app, press *Enter* to stop it from listening.
2. Close Chrome and shut down the web server.

You've now seen how to build a simple SignalR chat hub and two clients that interact with it: a web page using JavaScript and a console app using .NET. Now let's look at another feature of SignalR that can improve efficiency: streaming data.

Streaming data using SignalR

So far, we have seen how SignalR can broadcast structured messages to one or more clients. This works well with data that is relatively small and structured and exists completely at a point in time. But what about data that comes in parts over time?

Streams can be used for these scenarios. SignalR supports both service-to-client (downloading data from a stream) and client-to-service (uploading data to a stream).

To enable download streaming, a hub method must return `IAsyncEnumerable<T>` (only supported by C# 8 or later) or `ChannelReader<T>`.

To enable upload streaming, a hub method must accept a parameter of type `IAsyncEnumerable<T>` (only supported by C# 8 or later) or `ChannelReader<T>`.

Defining a hub for streaming

Let's add some streaming methods to see how they work in action:

1. In the `Northwind.Common` project, add a new file named `StockPrice.cs` and modify its contents to define a `record` for stock price data, as shown in the following code:

```
namespace Northwind.SignalR.Streams;
public record StockPrice(string Stock,
double Price);
```

2. Build the Northwind.SignalR.Service project to update its referenced projects.
3. In the Northwind.SignalR.Service project, in the Hubs folder, add a new class named StockPriceHub.cs, and modify its contents to define a hub with two streaming methods, as shown in the following code:

```
using Microsoft.AspNetCore.SignalR; // To
use Hub. using
System.Runtime.CompilerServices; // To use
[EnumeratorCancellation]. using
Northwind.SignalR.Streams; // To use
StockPrice. namespace
Northwind.SignalR.Service.Hubs; public
class StockPriceHub : Hub { public async
IAsyncEnumerable<StockPrice>
GetStockPriceUpdates( string stock,
[EnumeratorCancellation] CancellationToken
cancellationToken) { double currentPrice =
267.10; // Simulated initial price. for
(int i = 0; i < 10; i++) { // Check the
cancellation token regularly so that the
server will stop // producing items if the
client disconnects.
cancellationToken.ThrowIfCancellationRequested();
// Increment or decrement the current
price by a random amount. // The compiler
does not need the extra parentheses but it
// is clearer for humans if you put them
```

```

in. currentPrice +=
(Random.Shared.NextDouble() * 10.0) - 5.0;
StockPrice stockPrice = new(stock,
currentPrice); Console.WriteLine("[{0}]
{1} at {2:C}", DateTime.UtcNow,
stockPrice.Stock, stockPrice.Price); yield
return stockPrice; await Task.Delay(4000,
cancellationTokens); // milliseconds } }
public async Task
UploadStocks(IAsyncEnumerable<string>
stocks) { await foreach (string stock in
stocks) { Console.WriteLine($"Receiving
{stock} from client..."); } } }

```

4. In the `Northwind.SignalR.Service` project, in `Program.cs`, register the stock price hub after the statement that registers the chat hub, as shown in the following code:

```
app.MapHub<StockPriceHub>("/stockprice");
```

Creating a .NET console app client for streaming

Now, we can create a simple client to download a stream of data from the SignalR hub and upload a stream of data to the SignalR hub:

1. Use your preferred code editor to add a new project, as defined in the following list:
 - Project template: **Console Application** / console
 - Solution file and folder: `ModernServices`
 - Project file and folder:

```
Northwind.SignalR.Client.Console.Streams
```

5. In the `Northwind.SignalR.Client.Console.Streams` project file, treat warnings as errors, add a package reference for the ASP.NET Core SignalR client, add a project reference to `Northwind.Common`, and globally and statically import the `System.Console` class, as shown highlighted in the following markup:

```
<ItemGroup> <PackageReference  
Include="Microsoft.AspNetCore.SignalR.Client"  
/> </ItemGroup> <ItemGroup>  
<ProjectReference Include= "..  
\Northwind.Common\Northwind.Common.csproj"  
/> </ItemGroup>
```

6. In the `Northwind.SignalR.Client.Console.Streams` project, add a new class file named `Program.Methods.cs`, and modify its content to define static methods in the partial `Program` class to generate ten random four-letter stock codes asynchronously, as shown in the following code:

```
// Defined in the empty default namespace  
to merge with the auto- // generated  
partial Program class. partial class  
Program { static async  
IEnumerable<string> GetStocksAsync()  
{ for (int i = 0; i < 10; i++) { // Return  
a random four-letter stock code. yield  
return $"{AtoZ()} {AtoZ()} {AtoZ()}  
{AtoZ()}"; await  
Task.Delay(TimeSpan.FromSeconds(3)); } }  
static string AtoZ() { return  
char.ConvertFromUtf32(Random.Shared.Next(65,  
91)); } }
```

7. In the `Northwind.SignalR.Client.Console.Streams` project, in `Program.cs`, delete the existing statements. Import namespaces for working with SignalR as a client, and then add statements to prompt the user to enter a stock, create a hub connection, listen for received streams of stock prices, and then send an asynchronous stream of stocks to the service, as shown in the following code:

```
using Microsoft.AspNetCore.SignalR.Client;
// To use HubConnection. using
Northwind.SignalR.Streams; // To use
StockPrice. Write("Enter a stock (press
Enter for MSFT): "); string? stock =
ReadLine(); if
(string.IsNullOrEmpty(stock)) { stock =
"MSFT"; } HubConnection hubConnection =
new HubConnectionBuilder()
.WithUrl("https://localhost:5121/
stockprice") .Build(); await
hubConnection.StartAsync(); try {
CancellationTokenSource cts = new();
IAsyncEnumerable<StockPrice> stockPrices =
hubConnection.StreamAsync<StockPrice>(
"GetStockPriceUpdates", stock, cts.Token);
await foreach (StockPrice sp in
stockPrices) { WriteLine($"{sp.Stock} is
now {sp.Price:C}."); Write("Do you want to
cancel (y/n)? "); ConsoleKey key =
ReadKey().Key; if (key == ConsoleKey.Y) {
cts.Cancel(); } WriteLine(); } } catch
(Exception ex) {
WriteLine($"{ex.GetType()} says
{ex.Message}"); } WriteLine();
WriteLine("Streaming download
```

```
completed."); await
hubConnection.SendAsync("UploadStocks",
GetStocksAsync()); WriteLine("Uploading
stocks to service... (press ENTER to
stop."); ReadLine(); WriteLine("Ending
console app.");
```

Testing the streaming service and client

Finally, we can test the streaming data functionality:

1. Start the `Northwind.SignalR.Service` project website using the `https` profile without debugging.
2. Start the `Northwind.SignalR.Client.Console.Streams` console app without debugging.
3. Arrange the console windows for the ASP.NET Core MVC website and the client console app so that you can see both side by side.
4. In the client console app, press *Enter* to use the Microsoft stock code, as shown in the following output:

```
Enter a stock (press Enter for MSFT): MSFT
is now £265.00. Do you want to cancel (y/
n)?
```

5. In the website console window, wait for about ten seconds, and note that several stock prices have been generated in the service but have not yet been sent to the client, as shown in the following output:

```
info: Microsoft.Hosting.Lifetime[14] Now
listening on: https://localhost:5121 info:
Microsoft.Hosting.Lifetime[14] Now
listening on: http://localhost:5122 info:
```

```
Microsoft.Hosting.Lifetime[0] Application
started. Press Ctrl+C to shut down. info:
Microsoft.Hosting.Lifetime[0] Hosting
environment: Development info:
Microsoft.Hosting.Lifetime[0] Content root
path: C:\apps-services-
net10\ModernServices
\Northwind.SignalR.Service [12/09/2025
17:52:26] MSFT at £265.00 [12/09/2025
17:52:30] MSFT at £260.78 [12/09/2025
17:52:34] MSFT at £264.86 [12/09/2025
17:52:38] MSFT at £262.10
```

6. In the client console app, press *n* to receive the next updated price. Keep pressing *n* until the prices have been sent from the service and read by the client, and then press *y*. Note that a cancellation token is received by the SignalR service, so it stops, and the client now starts uploading stocks, as shown in the following output:

```
MSFT is now £260.78. Do you want to cancel
(y/n)? n MSFT is now £264.86. Do you want
to cancel (y/n)? n MSFT is now £262.10. Do
you want to cancel (y/n)? y
System.Threading.Tasks.TaskCanceledException
says A task was canceled. Streaming
download completed. Uploading stocks to
service... (press ENTER to stop.)
```

7. In the website console window, note that the random stock codes are received, as shown in the following output:

```
Receiving PJON from client... Receiving
VWJD from client... Receiving HMOJ from
```

```
client... Receiving QMQ from client...
```

8. Close both console windows.

Practicing and exploring

Test your knowledge and understanding by answering some questions and exploring this chapter's topics with deeper research.

Exercise 12.1 – Test your knowledge

Answer the following questions:

1. What transport does SignalR use, and which is the default?
2. What is a good practice for RPC method signature design?
3. What tool can you use to download the SignalR JavaScript library?
4. What happens if you send a SignalR message to a client with a connection ID that does not exist?
5. What are the benefits of separating a SignalR service from other ASP.NET Core components?

Exercise 12.2 – Explore topics

Use the links on the following page to learn more details about the topics covered in this chapter: <https://github.com/markjprice/apps-services-net10/blob/main/docs/book-links.md#chapter-12---broadcasting-real-time-communication-using-signalr>.

Summary

In this chapter, you learned about:

- The history of technologies before SignalR.

- The concepts and technologies that underpin SignalR.
- Implementing the chat functionality using SignalR, including building a hub hosted in a website project, and clients using JavaScript and a .NET console app.
- Downloading and uploading streams of data using SignalR.

In the next chapter, you will learn about GraphQL, another standard that enables client control over the data returned from a service.

Get This Book **UNLOCK NOW** and Exclusive

Scan the QR code
book by name, co



Search for this
ps on the page.

Note: *Keep your invoice handy. Purchases made directly from Packt don't require one.*

13

Combining Data Sources Using GraphQL

In this chapter, you will be introduced to GraphQL, a service technology that provides a more modern approach to combining data from various sources and a standard way to query that data.

This chapter will cover the following topics:

- Understanding GraphQL
- Building a service that supports GraphQL
- Defining GraphQL queries for EF Core models
- Building .NET clients for a GraphQL service
- Implementing GraphQL mutations
- Implementing GraphQL subscriptions

Understanding GraphQL

In [Chapter 10](#), *Building and Securing Minimal API Web Services*, you learned how to define a Web API service by mapping request path endpoints to lambda expressions or methods that return the response. Any parameters and the format of responses are under the control of the service. A client cannot ask for

exactly what they need or use more efficient data formats.

If you have read my book, *Real-World Web Development with .NET 10*, then you know that OData has a built-in query language for the client to control what data they want to be returned. However, OData has a rather old-fashioned approach and is tied to the HTTP standard, for example, using query strings in an HTTP request.

If you would prefer to use a more modern and flexible technology to combine and expose your data as a service, then a good alternative is **GraphQL**.

Like OData, GraphQL is a standard for describing your data and then querying it that gives the client control over exactly what they need. It was developed internally by Facebook in 2012 before being open sourced in 2015, and it is now managed by the GraphQL Foundation.

Some of the key benefits of GraphQL over OData are:

- GraphQL does not require HTTP because it is transport-agnostic, so you could use alternative transport protocols like WebSockets or TCP.
- GraphQL has more client libraries for different platforms than OData has.
- GraphQL has a single endpoint, usually simply `/graphql`.

GraphQL query document format

GraphQL uses its own document format for its queries, which is a bit like JSON, but GraphQL queries do not require commas between field names. This is shown in the following query, which requests some fields and related data for the product with an ID of 23:

```
{ product (productId: 23) { productId
  productName unitPrice supplier { companyName
  country } } }
```

The outer braces define the **selection set** for the root `Query` type. In simple

terms, this means: “From the root query, I want the `product` field.” GraphQL always starts at the root type; there is no equivalent of hitting an endpoint like `/products/23`.

`product (productId: 23): product` is a field on the root `Query` type. It accepts an argument called `productId`, and we are passing the value `23`. Here we’re asking, “Give me the product whose ID is 23.” This is a function call, not a REST-style URL. GraphQL fields can take arguments at any level, not just at the top.

The inner block is the selection set for the returned `Product` object. GraphQL never returns an object unless you explicitly ask for its fields. You are requesting exactly three scalar fields: `productId`, `productName`, and `unitPrice`. Nothing else will be returned. No timestamps, no descriptions, no internal IDs, no junk. This is one of GraphQL’s big wins over REST.

Then we have a nested object, `supplier`. Because `supplier` is an object, you must specify its fields too. You are asking for exactly: `companyName` and `country`. Again, nothing else comes back. No address, no phone number, no supplier ID.

GraphQL guarantees that the shape of the response mirrors the shape of the query. So the JSON response will look roughly like the following:

```
{ "data": { "product": { "productId": 23,
  "productName": "Tunnbröd", "unitPrice": 9.00,
  "supplier": { "companyName": "PB Knäckebröd
  AB", "country": "Sweden" } } } }
```

Even if the server uses a SQL database, a NoSQL store, or something else entirely, the client never sees that. The query defines the output.

Warning! This does not imply joins or SQL. The server resolver decides how to fetch data. This does

not guarantee performance. A badly written resolver can still cause slow queries. This does not mean the client can access everything. The schema strictly controls what fields exist. If `unitPrice` or `supplier` is not defined in the schema, this query fails validation before execution.

The official media type for GraphQL query documents is `application/graphql`.

Requesting fields

The most basic GraphQL query requests one or more fields from a type, for example, requesting three fields for each `customer` entity, as shown in the following code:

```
# The query keyword is optional. Comments are  
prefixed with #. query { customer { customerId  
  companyName country } }
```

The response is in the JSON format, for example, an array of `customer` objects, as shown in the following document:

```
{ "data": [ { "customerId": "ALFKI",  
  "companyName": "Alfreds Futterkiste",  
  "country": "Germany" }, ... ] }
```

Specifying filters and arguments

As well as requesting one or more fields from a type, a more complex query might specify filters. With an HTTP or REST-style API, the caller is limited to only passing parameters when the API predefines them. With GraphQL, you can

set parameters anywhere in the query, for example, filtering orders by order date and by the country of the customer who made the order, as shown in the following code:

```
query GetOrdersByDateAndCountry {  
  order(orderDate: "23/04/1998") { orderId  
    orderDate customer(country: "UK") {  
      companyName country } } }
```

Note that although GraphQL uses camelCase for entity, property, and parameter names, you should use PascalCase for query names.

You might want to pass values for named parameters instead of hardcoding them, as shown in the following code:

```
query GetOrdersByDateAndCountry($country:  
String, $orderDate: String) { order(orderDate:  
$orderDate) { orderId orderDate  
  customer(country: $country) { companyName  
    country } } }
```

You can learn more about the GraphQL query language at the following link: <https://graphql.org/learn/queries/>.

Understanding other GraphQL capabilities

As well as queries, other standard GraphQL features are mutations and subscriptions:

- **Mutations** enable you to create, update, and delete resources.
- **Subscriptions** enable a client to get notified when resources change.

They work best with additional communication technologies like WebSockets.

You will see these features later in this chapter in the sections titled, *Implementing GraphQL mutations* and *Implementing GraphQL subscriptions*.

Unfortunately, Microsoft does not provide an implementation of GraphQL in .NET, so we will have to use a third-party implementation.

Understanding the ChilliCream GraphQL platform

GraphQL.NET is one of the most popular platforms for implementing GraphQL with .NET. In my opinion, GraphQL.NET requires too much configuration for even the most basic example, and the documentation is frustrating. I have a feeling that, although it is powerful, it implements the GraphQL specification in a one-to-one manner instead of rethinking how a .NET platform could make it easier to get started.

You can learn about GraphQL.NET at the following link: <https://graphql-dotnet.github.io/>.

For this book, I looked for alternatives, and I found exactly what I was looking for. ChilliCream is a company that has created a whole platform to work with GraphQL. Their platform consists of a few products, as described in the following list:

- **Hot Chocolate** enables you to create GraphQL services for .NET. In this chapter we will use this product to build a GraphQL service hosted in an ASP.NET Core project.
- **Nitro** (previously known as **Banana Cake Pop**) enables you to run queries and explore a GraphQL endpoint using a Monaco-based GraphQL IDE. In this chapter we will use this product to try out our

GraphQL service and explore its capabilities without needing to build a proper client.

- **Strawberry Shake** enables you to create GraphQL clients for .NET. In this chapter we will use this product to build a strongly-typed client to call our GraphQL service.
- **Green Donut** enables better performance when loading data. This product is beyond the scope of this book.

Unlike some other packages that can be used to add support for GraphQL, ChilliCream packages are designed to be as easy to implement as possible, using conventions and simple POCO classes instead of complex types and special schemas. They work in a similar way to how Microsoft might have built a GraphQL platform for .NET, with sensible defaults and conventions rather than lots of boilerplate code and configuration.

As ChilliCream says on its home page, *“We at ChilliCream build the ultimate GraphQL platform. Most of our code is open-source and will forever remain open-source.”*

The GitHub repository for Hot Chocolate is at the following link: <https://github.com/ChilliCream/hotchocolate>.

Enough theory, let's get going with a practical example of a service that implements GraphQL.

Building a service that supports GraphQL

There is no `dotnet new` project template for GraphQL, so we will use the **ASP.NET Core Empty** project template. Even though GraphQL does not have to be hosted on a web server because it is not tied to HTTP, it is a sensible choice to get started. We will then add a package reference for GraphQL

support:

1. Use your preferred code editor to add a new project, as defined in the following list:

- Project template: **ASP.NET Core Empty** / web
- Solution file and folder: `ModernServices`
- Project file and folder:
`Northwind.GraphQL.Service`
- Other Visual Studio options:
- **Configure for HTTPS**: Selected
- **Enable container support**: Cleared
- **Do not use top-level statements**: Cleared
- **Use the .dev.localhost TLD in the application URL**:
Cleared
- **Enlist in Aspire orchestration**: Cleared

11. In the project file, add a package reference for Hot Chocolate hosted in ASP.NET Core, as shown in the following markup:

```
<ItemGroup> <PackageReference  
Include="HotChocolate.AspNetCore" /> </  
ItemGroup>
```

12. In the project file, treat warnings as errors, as highlighted in the following markup:

```
<PropertyGroup> <TargetFramework>net10.0</  
TargetFramework> <Nullable>enable</  
Nullable> <ImplicitUsings>enable</  
ImplicitUsings>  
<TreatWarningsAsErrors>true</  
TreatWarningsAsErrors> </PropertyGroup>
```

13. In the `Properties` folder, in `launchSettings.json`, for the `https` profile, modify the `applicationUrl` to use port 5131 for `https` and port 5132 for `http`. Then, add a `launchUrl` of `graphql`, as highlighted in the following configuration:

```
"https" :{ "commandName": "Project",
"dotnetRunMessages": true,
"launchBrowser": true, "launchUrl"
"graphql":, "applicationUrl" "https://
localhost:5131;http://localhost:5132":,
"environmentVariables": {
"ASPNETCORE_ENVIRONMENT": "Development" }
```

14. Build the `Northwind.GraphQL.Service` project.

Defining the GraphQL schema for Hello World

The first task is to define what we want to expose as GraphQL models in the web service.

Let's define a GraphQL query for the most basic `Hello World` example, which will respond with plain text when a request for a greeting is made:

1. In the `Northwind.GraphQL.Service` project/folder, add a class file named `Query.cs`.
2. Modify the class to have a method named `GetGreeting` that returns the plain text `"Hello, World!"`, as shown in the following code:

```
namespace Northwind.GraphQL.Service;
public class Query { public string
GetGreeting() => "Hello, World!"; }
```

3. In `Program.cs`, import the namespace where we defined the `Query` class, as shown in the following code:

```
using Northwind.GraphQL.Service; // To use
Query.
```

4. In the section to configure services, after the call to `CreateBuilder`, add a statement to add GraphQL server-side support, and add the query type to the collection of registered services, as shown in the following code:

```
builder.Services .AddGraphQLServer()
    .AddQueryType<Query>();
```

5. Modify the statement that maps a `GET` request to return a more useful plain text message, as shown in the following code:

```
app.MapGet("/", () => "Navigate to:
https://localhost:5131/graphql");
```

6. In the section to configure the HTTP pipeline, before the call to `Run`, add a statement to map GraphQL as an endpoint, as shown in the following code:

```
app.MapGraphQL();
```

7. Start the `Northwind.GraphQL.Service` web project, using the `https` profile without debugging:
 - If you are using Visual Studio, then select the **https** profile, start the project without debugging, and note that

the browser starts automatically.

- If you are using VS Code, then enter the command `dotnet run --launch-profile https`, start Chrome manually, and navigate to `https://localhost:5131/graphql`.

10. Note the **Nitro** user interface, and then click the **Create Document** button.

11. In the top-right corner, click the **Connection Settings** button, as shown in *Figure 13.1*:

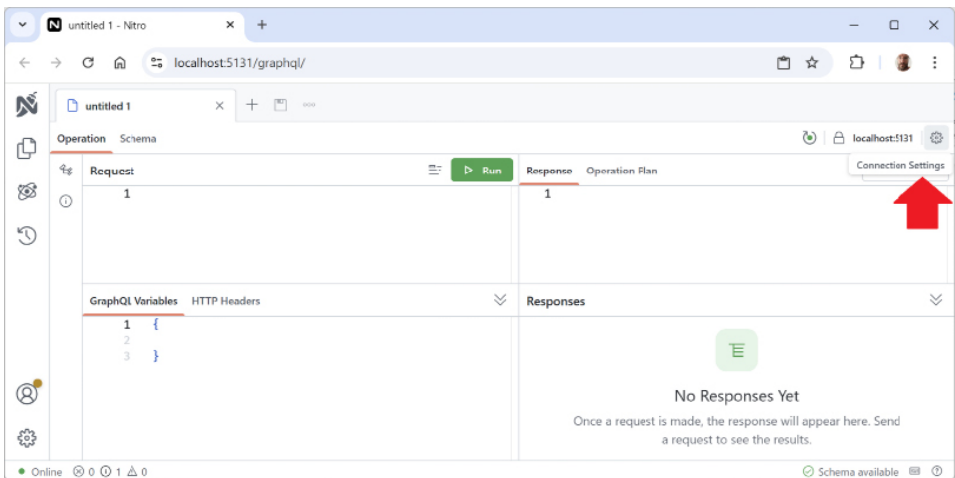


Figure 13.1: A new Nitro document and connection settings button

1. In **Connection Settings**, review the endpoints and other configuration, and then click **Cancel**, as shown in *Figure 13.2*:

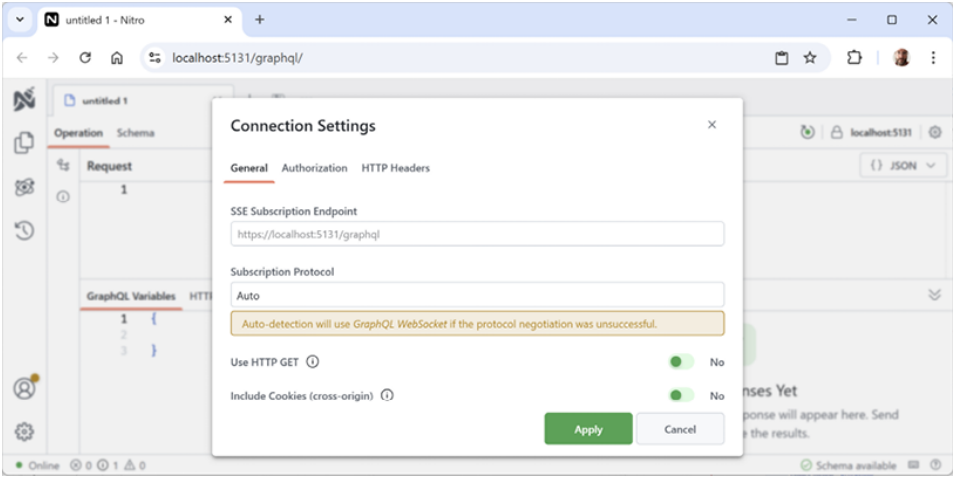


Figure 13.2: Reviewing the Nitro connection settings

1. At the top of the **untitled 1** document, click the **Schema** tab.
2. In the **Schema** tab, note the `Query` type is a special type that defines the entry point of every GraphQL query, and it has a field named `greeting` that returns a `String!` value. The exclamation mark indicates that the value will *not* be `null`.
3. Click the **SDL** tab, and note there is only one type defined, the special `Query` object with its `greeting` field, which is a non-null `String` value, as shown in the following code:

```
type Query { greeting: String! }
```

Writing and executing GraphQL queries

Now that we know the schema, we can write and run a query:

1. In **Nitro**, in the **untitled 1** document, click the **Operation** tab.
2. On the left-hand side, type an open curly brace, `{`, and note that a close curly brace, `}`, is written for you.

3. Type the letter `g`, and note that the autocomplete shows it recognizes the greeting field, as shown in *Figure 13.3*:

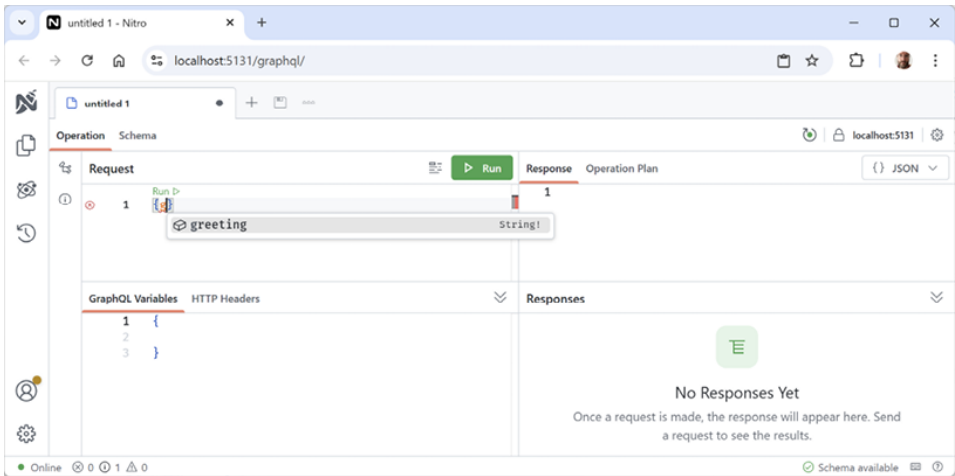


Figure 13.3: Autocomplete for the greeting field

1. Press *Enter* to accept the autocomplete suggestion.
2. Click the **Run** button and note the response, as shown in the following output:

```
{ "data": { "greeting": "Hello, World!" } }
```

3. Close Chrome, and shut down the web server.

Naming GraphQL queries aka operations

The query that we wrote was unnamed. We could also have created it as a named query, as shown in the following code:

```
query QueryNameGoesHere { greeting }
```

Named queries allow clients to identify queries and responses for telemetry purposes, for example, when hosting in Microsoft Azure cloud services and monitoring using Application Insights.

Understanding field conventions

The C# method we created in the `Query` class was named `GetGreeting`, but when querying it, we used `greeting`. The `Get` prefix on method names that represent fields in GraphQL is optional. Let's see some more examples:

1. In `Query.cs`, add two more methods without the `Get` prefix, as highlighted in the following code:

```
namespace Northwind.GraphQL.Service;
public class Query { public string
GetGreeting() => "Hello, World!"; public
string Farewell() => "Ciao! Ciao!"; public
int RollTheDie() => Random.Shared.Next(1,
7); }
```

2. Start the `Northwind.GraphQL.Service` project, using the `https` profile without debugging.
3. Click the **Schema** tab, and note the updated schema, as shown in the following code:

```
type Query { greeting: String! farewell:
String! rollTheDie: Int! }
```

C# methods use PascalCase. GraphQL fields use camelCase.

1. Click the **Operation** tab, and in the **Request** box, modify the query to

specify a name and request the `rollTheDie` field, as shown in the following code:

```
query GetNumber { rollTheDie }
```

2. Click **Run** again multiple times. Note that the responses contain a random number between 1 and 6, and a history of requests and responses is stored for the current browser session, as shown in *Figure 13.4*:

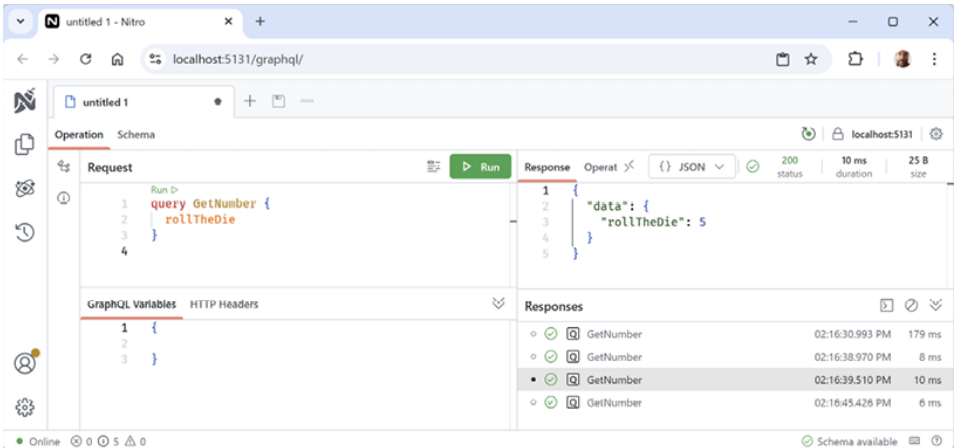


Figure 13.4: Executing a named query and the history of requests and responses

1. Close Chrome, and shut down the web server.

Now that you have seen how to define some simple GraphQL schemas and then write and execute some queries that call them, let's get more practical and look at how to define some GraphQL schemas for EF Core models so that a client can query a live database.

Defining GraphQL queries for EF Core models

Now that we have a basic GraphQL service operating successfully, let's extend

it to enable querying the Northwind database.

Adding support for EF Core

We must add another Hot Chocolate package to allow easy dependency integration of our EF Core database context with GraphQL query classes:

1. Add a package reference for Hot Chocolate integration with EF Core and a project reference to the Northwind database context project, as shown highlighted in the following markup:

```
<ItemGroup> <PackageReference  
  Include="HotChocolate.AspNetCore" />  
<PackageReference  
  Include="HotChocolate.Data.EntityFramework"  
/> </ItemGroup> <ItemGroup>  
  <ProjectReference Include= "..\..  
    \ModernApps\Northwind.DataContext  
    \Northwind.DataContext.csproj" /> </  
ItemGroup>
```

The path to the project must not have a line break. All Hot Chocolate packages should have the same version number.

1. Build the `Northwind.GraphQLService` project.
2. In `Program.cs`, import the namespace to work with our EF Core model for the Northwind database, as shown in the following code:

```
using Northwind.EntityModels; // To use  
AddNorthwindContext method.
```

3. Add a statement after the `CreateBuilder` method to register the

Northwind database context class, and add a statement after adding the GraphQL server support to register the `NorthwindContext` class for dependency injection, as highlighted in the following code:

```
builder.Services.AddNorthwindContext();  
builder.Services.AddGraphQLServer()  
    .RegisterDbContext<NorthwindContext>()  
    .AddQueryType<Query>();
```

4. In `Query.cs`, add statements to define an object graph type that includes queries to return a list of categories, a single category, products for a category, products with a minimum unit price, and all products, as highlighted in the following code:

```
using Microsoft.EntityFrameworkCore; // To  
use Include method. using  
Northwind.EntityModels; // To use  
NorthwindContext. namespace  
Northwind.GraphQL.Service; public class  
Query { public string GetGreeting() =>  
    "Hello, World!"; public string Farewell()  
=> "Ciao! Ciao!"; public int RollTheDie()  
=> Random.Shared.Next(1, 7); public  
IQueryable<Category>  
GetCategories(NorthwindContext db) =>  
    db.Categories.Include(c => c.Products);  
public Category?  
GetCategory(NorthwindContext db, int  
categoryId) { Category? category =  
    db.Categories.Find(categoryId); if  
    (category == null) return null;  
    db.Entry(category).Collection(c =>  
        c.Products).Load(); return category; }
```

```

public IQueryable<Product>
GetProducts(NorthwindContext db) =>
db.Products.Include(p => p.Category);
public IQueryable<Product>
GetProductsInCategory( NorthwindContext
db, int categoryId) => db.Products.Where(p
=> p.CategoryId == categoryId); public
IQueryable<Product>
GetProductsByUnitPrice( NorthwindContext
db, decimal minimumUnitPrice) =>
db.Products.Where(p => p.UnitPrice >=
minimumUnitPrice); }

```

Exploring GraphQL queries with Northwind

Now we can test writing GraphQL queries for the categories and products in the Northwind database:

1. If your database server is not running, for example, because you are hosting it in Docker, a virtual machine, or the cloud, then make sure to start it.
2. Start the `Northwind.GraphQL.Service` project, using the `https` profile without debugging.
3. In **Nitro**, click the **+** to open a new tab.
4. Click the **Schema** tab, then the **SDL** tab, and note the query and type definitions for `Category`, as partially shown in *Figure 13.5*:

2. Click the **Operation** tab, and write a named query to request the ID, name, and description fields for all categories, as shown in the following markup:

```
query AllCategories { categories {  
  categoryId categoryName description } }
```

3. Click **Run**, and note the response, as shown in *Figure 13.6* and the following partial output:

```
{ "data": { "categories": [ {  
  "categoryId": 1, "categoryName":  
  "Beverages", "description": "Soft drinks,  
  coffees, teas, beers, and ales" }, {  
  "categoryId": 2, "categoryName":  
  "Condiments", "description": "Sweet and  
  savory sauces, relishes, spreads, and  
  seasonings" }, ...
```

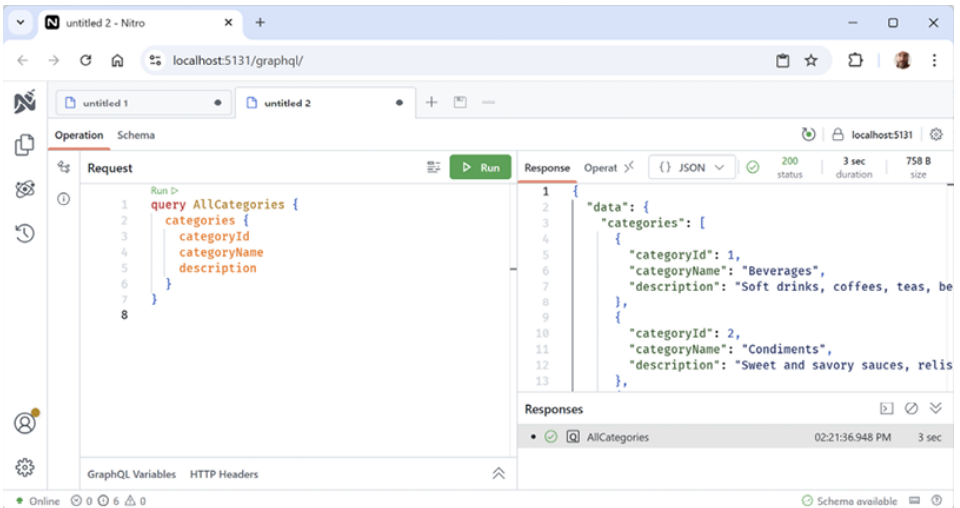


Figure 13.6: Getting all categories

1. Click the + to open a tab for a new document, and write a query to request the category with ID 2, including the ID, name, and price of its products, as shown in the following markup:

```
query Condiments { category (categoryId:  
2) { categoryId categoryName products {  
productId productName unitPrice } } }
```

Make sure that the I in categoryId is uppercase.

1. Click **Run**, and note the response, as shown in the following partial output:

```
{ "data": { "category": { "categoryId": 2,  
"categoryName": "Condiments", "products":  
[ { "productId": 3, "productName":  
"Aniseed Syrup", "unitPrice": 10 }, {  
"productId": 4, "productName": "Chef  
Anton's Cajun Seasoning", "unitPrice": 22  
}, ...
```

2. In the GraphQL web service command prompt or terminal, note the SQL statements executed for this query, as shown in the following output:

```
info:  
Microsoft.EntityFrameworkCore.Database.Command [20  
Executed DbCommand (68ms)  
[Parameters=[@__p_0='?' (DbType = Int32)],  
CommandType='Text', CommandTimeout='30']
```

```

SELECT TOP (1) [c].[CategoryId], [c].
[CategoryName], [c].[Description], [c].
[Picture] FROM [Categories] AS [c] WHERE
[c].[CategoryId] = @__p_0 info:
Microsoft.EntityFrameworkCore.Database.Command
Executed DbCommand (5ms)
[Parameters=[@__p_0='?' (DbType = Int32)],
CommandType='Text', CommandTimeout='30']
SELECT [p].[ProductId], [p].[CategoryId],
[p].[Discontinued], [p].[ProductName],
[p].[QuantityPerUnit], [p].[ReorderLevel],
[p].[SupplierId], [p].[UnitPrice], [p].
[UnitsInStock], [p].[UnitsOnOrder] FROM
[Products] AS [p] WHERE [p].[CategoryId] =
@__p_0

```

Although the GraphQL query did not need the picture of each category and only needed the ID, name, and unit price, the dynamically-generated queries from EF Core returned all properties.

1. Click the **+** tab to open a new tab, and write a query to request the ID, name, and units in stock of the products in the category with ID 1, as shown in the following markup:

```

query BeverageProducts {
  productsInCategory (categoryId: 1) {
    productId productName unitsInStock } }

```

2. Click **Run**, and note the response, as shown in the following partial output:

```
{ "data": { "productsInCategory": [ {  
  "productId": 1, "productName": "Chai",  
  "unitsInStock": 39 }, { "productId": 2,  
  "productName": "Chang", "unitsInStock": 17  
}, ...
```

3. Click the + tab to open a new tab, and write a query to request the ID, name, and units in stock of products, along with their category names, as shown in the following markup:

```
query ProductsWithCategoryNames { products  
  { productId productName category {  
    categoryName } unitsInStock } }
```

4. Click **Run**, and note the response, as shown in the following partial output:

```
{ "data": { "products": [ { "productId":  
1, "productName": "Chai", "category": {  
  "categoryName": "Beverages" },  
  "unitsInStock": 39 }, { "productId": 2,  
  "productName": "Chang", "category": {  
    "categoryName": "Beverages" },  
  "unitsInStock": 17 }, ...
```

5. Click the + tab to open a new tab, and write a query to request the ID and name of a category, select the category by specifying its category ID, and include the ID and name for each of its products. The ID of the category will be set using a variable, as shown in the following markup:

```
query CategoryAndItsProducts($id: Int!){
```



```
category(categoryId: $id) { categoryId
categoryName products { productId
productName } } }
```

6. In the **GraphQL Variables** section, define a value for the variable, as shown in the following code and in *Figure 13.7*:

```
{ "id": 1 }
```

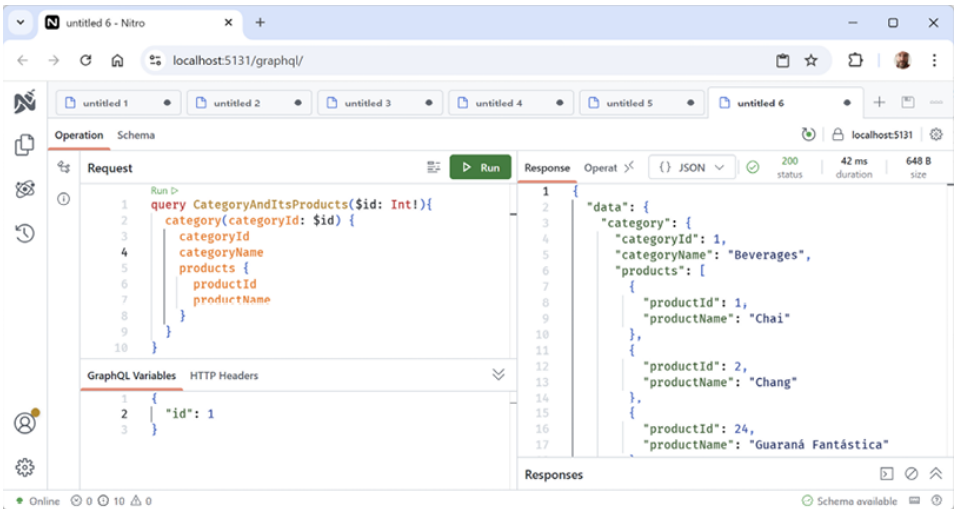


Figure 13.7: Executing a GraphQL query with a variable

1. Click **Run**, and note the response, as shown in the following partial output:

```
{ "data": { "category": { "categoryId": 1,
"categoryName": "Beverages", "products": [
{ "productId": 1, "productName": "Chai" },
{ "productId": 2, "productName": "Chang"
}, ...
```

2. Click the **+** tab to open a new tab, write a query to request the ID and name of a category, select the category by specifying its category ID, and include the ID and name for each of its products. The ID of the category will be set using a variable, as shown in the following markup:

```
query ProductsWithMinimumPrice($unitPrice:
Decimal!) {
  productsByUnitPrice(minimumUnitPrice:
$unitPrice) { productId productName
unitPrice } }
```

3. In the **GraphQL Variables** section, define a value for the variable, as shown in the following code:

```
{ "unitPrice": 100 }
```

4. Click **Run**, and note the response, as shown in the following output:

```
{ "data": { "productsByUnitPrice": [ {
  "productId": 29, "productName": "Thüringer
Rostbratwurst", "unitPrice": 123.79 }, {
  "productId": 38, "productName": "Côte de
Blaye", "unitPrice": 263.5 } ] } }
```

5. Close Chrome, and shut down the web server.

Implementing paging support

When we request products using the `GetProducts` method (which maps to the `products` query), all 77 products are returned. Now let's add paging support:

1. In `Query.cs`, add statements to define a query to return all products, using paging, and note that its implementation is the same as the query for products without paging, but it is decorated with the `[UsePaging]` attribute, as shown in the following code:

```
[UsePaging] public IQueryable<Product>
GetProductsWithPaging(NorthwindContext db)
=> db.Products.Include(p => p.Category);
```

Good practice: The `[UsePaging]`, `[UseFiltering]`, and `[UseSorting]` attributes must be decorated onto a query method that returns `IQueryable<T>`, allowing GraphQL to dynamically configure the LINQ query before executing it against the data store.

1. Start the `Northwind.GraphQL.Service` project, using the `https` profile without debugging.
2. In **Nitro**, click the **+** to open a new tab.
3. Click the **Schema | SDL** tab, and note the queries named `products` (without paging) and `productsWithPaging`, which documents how to use the query to request a page of products, as shown in the following output:

```
products: [Product!]!
productsInCategory(categoryId: Int!):
  [Product!]!
productsByUnitPrice(minimumUnitPrice:
  Decimal!): [Product!]! productsWithPaging(
  """ Returns the first _n_ elements from
```

```
the list. """ first: Int """ Returns the
elements in the list that come after the
specified cursor. """ after: String """
Returns the last _n_ elements from the
list. """ last: Int """ Returns the
elements in the list that come before the
specified cursor. """ before: String ):
ProductsWithPagingConnection
```

4. Click the **Operation** tab, and write a named query to request the first page of 10 products, as shown in the following markup:

```
query FirstTenProducts {
  productsWithPaging(first: 10) { pageInfo {
    hasPreviousPage hasNextPage startCursor
    endCursor } nodes { productId productName
  } } }
```

5. Click **Run**, and note the response, including the `pageInfo` section, which tells us that there is another page of products and that the cursor for this page ranges from `MA==` to `OQ==`, as shown in the following partial output:

```
{ "data": { "productsWithPaging": {
  "pageInfo": { "hasPreviousPage": false,
    "hasNextPage": true, "startCursor":
    "MA==", "endCursor": "OQ==" }, "nodes": [
    { "productId": 1, "productName": "Chai" },
    ... { "productId": 10, "productName":
    "Ikura" } ] } } }
```

6. Click the **+** to open a new tab, and write a named query to request the

second page of 10 products, specifying that we want the cursor that starts after `OQ==`, as shown in the following markup:

```
query SecondTenProducts {
  productsWithPaging(after: "OQ==") {
    pageInfo { hasPreviousPage hasNextPage
      startCursor endCursor } nodes { productId
      productName } } }
```

7. Click **Run**, and note the response, including the `pageInfo` section, which tells us that there is another page of products and the cursor for this page ranges from `MTA=` to `MTk=`, as shown in the following partial output:

```
{ "data": { "productsWithPaging": {
  "pageInfo": { "hasPreviousPage": true,
    "hasNextPage": true, "startCursor":
    "MTA=", "endCursor": "MTk=" }, "nodes": [
    { "productId": 11, "productName": "Queso
    Cabrales" }, ... { "productId": 20,
    "productName": "Sir Rodney's Marmalade" }
  ] } } }
```

8. Close Chrome, and shut down the web server.

Implementing filtering support

When we explored queries earlier in this chapter, we predefined some queries with parameters, for example, a query that returns all the products in a category by passing a `categoryId` parameter.

However, what if you do not know ahead of time which filtering you want to perform?

Let's add filtering support to our GraphQL queries:

1. In `Program.cs`, add a call to `AddFiltering` after calling `AddGraphQLServer`, as highlighted in the following code:

```
builder.Services .AddGraphQLServer()  
                .AddFiltering()  
                .RegisterDbContext<NorthwindContext>()  
                .AddQueryType<Query>();
```

2. In `Query.cs`, decorate the `GetProducts` method with the `[UseFiltering]` attribute, as highlighted in the following code:

```
[UseFiltering] public IQueryable<Product>  
GetProducts(NorthwindContext db) =>  
db.Products.Include(p => p.Category);
```

3. Start the `Northwind.GraphQL.Service` project, using the `https` profile without debugging.
4. In **Nitro**, click the **+** to open a new tab.
5. Click **Scheme Definition**, and note that the `products` query now accepts a filter input, as shown in the following markup:

```
products(where: ProductFilterInput):  
[Product!]!
```

6. Scroll down the schema definition to find `ProductFilterInput`, and note the filtering options include Boolean operators like `and` and `or`, as well as field filters like `IntOperationFilterInput` and `StringOperationFilterInput`, as shown in the following markup:

```

input ProductFilterInput { and:
  [ProductFilterInput!] or:
  [ProductFilterInput!] productId:
  IntOperationFilterInput productName:
  StringOperationFilterInput supplierId:
  IntOperationFilterInput categoryId:
  IntOperationFilterInput quantityPerUnit:
  StringOperationFilterInput unitPrice:
  DecimalOperationFilterInput unitsInStock:
  ShortOperationFilterInput unitsOnOrder:
  ShortOperationFilterInput reorderLevel:
  ShortOperationFilterInput discontinued:
  BooleanOperationFilterInput category:
  CategoryFilterInput orderDetails:
  ListFilterInputTypeOfOrderDetailFilterInput
  supplier: SupplierFilterInput }

```

7. Scroll down the schema definition to find

`IntOperationFilterInput` and

`StringOperationFilterInput`, and note the operations you can use with them, like `equals (eq)`, `not equals (neq)`, in an array (`in`), `greater than (gt)`, `contains`, and `startsWith`, as shown in the following markup:

```

input IntOperationFilterInput { eq: Int
  neq: Int in: [Int] nin: [Int] gt: Int ng: Int
  gte: Int ngte: Int lt: Int nlt: Int
  lte: Int nlte: Int } input
StringOperationFilterInput { and:
  [StringOperationFilterInput!] or:
  [StringOperationFilterInput!] eq: String
  neq: String contains: String ncontains:
  String in: [String] nin: [String]

```

```
startsWith: String nstartsWith: String  
endsWith: String nendsWith: String }
```

8. Click **Operations**, and then write a named query to request products with more than 120 units in stock, as shown in the following markup:

```
query ProductsWithMoreThan120InStock {  
  products(where: { unitsInStock: { gt: 120  
    } }) { productId productName unitsInStock  
    } }
```

9. Click **Run**, and note the response, as shown in the following output:

```
{ "data": { "products": [ { "productId":  
40, "productName": "Boston Crab Meat",  
"unitsInStock": 123 }, { "productId": 75,  
"productName": "Rhönbräu Klosterbier",  
"unitsInStock": 125 } ] } }
```

10. Click the + to open a new tab, click **Operation**, and then write a named query to request products whose names start with Cha, as shown in the following markup:

```
query ProductNamesCha { products(where: {  
  productName: { startsWith: "Cha" } }) {  
  productId productName } }
```

11. Click **Run**, and note the response, as shown in the following output:

```
{ "data": { "products": [ { "productId":  
1, "productName": "Chai" }, { "productId":
```



```
2, "productName": "Chang" }, {  
  "productId": 39, "productName":  
  "Chartreuse verte" } ] } }
```

12. In the GraphQL service command prompt or terminal, note the EF Core-generated SQL filters using parameters, as shown in the following output:

```
info:  
Microsoft.EntityFrameworkCore.Database.Command  
Executed DbCommand (2ms)  
[Parameters=[@__p_0_rewritten='?' (Size =  
40)], CommandType='Text',  
CommandTimeout='30'] SELECT [p].  
[ProductId], [p].[CategoryId], [p].  
[Discontinued], [p].[ProductName], [p].  
[QuantityPerUnit], [p].[ReorderLevel],  
[p].[SupplierId], [p].[UnitPrice], [p].  
[UnitsInStock], [p].[UnitsOnOrder], [c].  
[CategoryId], [c].[CategoryName], [c].  
[Description], [c].[Picture] FROM  
[Products] AS [p] LEFT JOIN [Categories]  
AS [c] ON [p].[CategoryId] = [c].  
[CategoryId] WHERE [p].[ProductName] LIKE  
@__p_0_rewritten ESCAPE N'\'
```

13. Close Chrome, and shut down the web server.

Implementing sorting support

To enable sorting with a GraphQL service, call the `AddSorting` method, as highlighted in the following code:

```
builder.Services .AddGraphQLServer()
```

```
.AddFiltering() .AddSorting()  
.RegisterDbContext<NorthwindContext>()  
.AddQueryType<Query>();
```

Then, decorate a query method that returns `IQueryable<T>` with the `[UseSorting]` attribute, as highlighted in the following code:

```
[UseFiltering] [UseSorting] public  
IQueryable<Product>  
GetProducts(NorthwindContext db) =>  
db.Products.Include(p => p.Category);
```

In a query, apply one or more sort orders, as shown in the following code:

```
query ProductsSortedByMostExpensive {  
  products(order: [ { unitPrice: DESC } ]) {  
    productId productName unitPrice } } }
```

`SortEnumType` has two values, as shown in the following code:

```
enum SortEnumType { ASC DESC }
```

I will leave adding sorting capabilities to your GraphQL service to you.

Now that we have built a GraphQL service with lots of interesting capabilities, let's see how you would create clients that can call it.

Building .NET clients for a GraphQL service

Now that we have explored some queries with the **Nitro** tool, let's see how a

client could call the GraphQL service. Although the **Nitro** tool is convenient, it runs in the same domain as the service, so some issues might not become apparent until we create a separate client.

Choosing GraphQL request formats

Most GraphQL services process `GET` and `POST` requests in either the `application/graphql` or `application/json` media formats. An `application/graphql` request would only contain a query document. The benefit of using `application/json` is that, as well as the query document, you can specify operations when you have more than one, and define and set variables, as shown in the following code:

```
{ "query": "...", "operationName": "...",  
  "variables": { "variable1": "value1", ... } }
```

We will use the `application/json` media format so that we can pass variables and their values.

Understanding the GraphQL response format

A GraphQL service should return a JSON document containing the expected data object and maybe some errors in an array, with the following structure:

```
{ "data": { ... }, "errors": [ ... ] }
```

The `errors` array should only be in the document if there are errors.

Using REST Client as a GraphQL client

Before we write code as a client to the GraphQL service, it would be good to

test it with your code editor's .http file support. This is so that if our .NET client app does not work, we know the problem is in our client code rather than the service:

1. If you are using VS Code and have not already installed REST Client by Huachao Mao (humao.rest-client), then install it now.
2. In your preferred code editor, start the Northwind.GraphQL.Service project web service, using the https profile without debugging, and leave it running.
3. In your code editor, in the HttpRequests folder, create a file named graphql-queries.http, and modify its contents to contain a request to get products in the seafood category, as shown in the following code:

```
### Configure a variable for the GraphQL
service base address. @base_address =
https://localhost:5131/graphql ### Get all
products in the specified category. POST
{{base_address}} Content-Type:
application/json { "query" :
"{productsInCategory(categoryId:8)
{productId productName unitsInStock}}" }
```

4. Send the query request, and note the response, as shown in *Figure 13.8*:

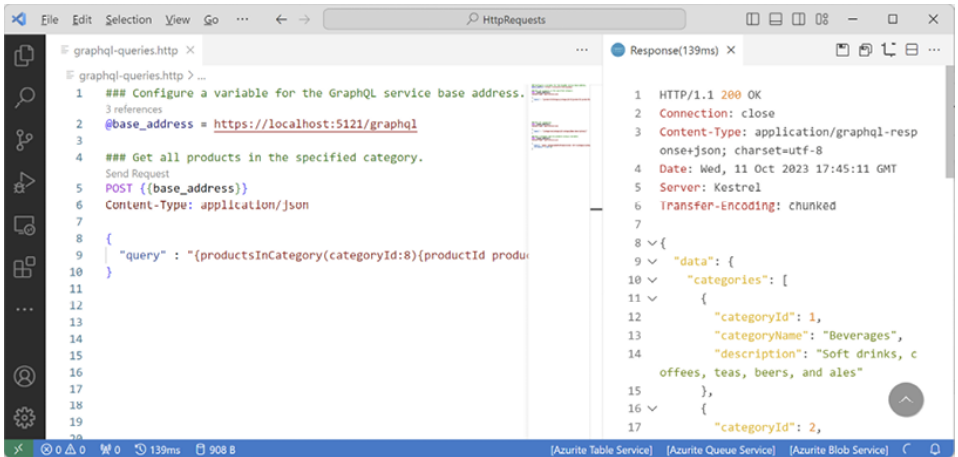


Figure 13.8: Requesting seafood products using REST Client

1. Add a query to make a request to get the ID, name, and description of all categories, as shown in the following code:

```
### Get all categories. POST
{{base_address}} Content-Type:
application/json { "query" :
"{categories{categoryId categoryName
description}}" }
```

2. Send the query request, and note the response contains the eight categories in a `data` property.
3. In the query document, change `categoryId` to `id`.
4. Send the query request, and note the response contains an `errors` array, as shown in the following response:

```
Response time: 60 ms Status code:
BadRequest (400) Alt-Svc: h3=":5131";
ma=86400 Transfer-Encoding: chunked Date:
Tue, 06 Jun 2025 16:35:18 GMT Server:
```

```
Kestrel Content-Type: application/graphql-response+json; charset=utf-8 Content-Length: 338
```

```
-----  
Content: { "errors": [ { "message": "The  
field `id` does not exist on the type  
`Category`.", "locations": [ { "line": 1,  
"column": 13 } ], "path": [ "categories"  
], "extensions": { "type": "Category",  
"field": "id", "responseName": "id",  
"specifiedBy": "http://spec.graphql.org/  
October2021/#sec-Field-Selections-on-  
Objects-Interfaces-and-Unions-Types" } } ]  
}
```

5. In the query document, change `id` back to `categoryId`.
6. Add a query to the request to get the ID and name of a category specified by a parameter for its ID, along with the ID and name of each of its products, as shown in the following code:

```
### Get a category and its products using  
a variable. POST {{base_address}} Content-  
Type: application/json { "query": "query  
categoryAndItsProducts($id: Int!)  
{category(categoryId: $id){categoryId  
categoryName products{productId  
productName}}}", "variables": {"id":1} }
```

7. Send the query request, and note the response contains category 1, Beverages, with its products in a `data` property.
8. Change the ID to 4, send the request, and note the response contains category 4, Dairy Products, with its products in a `data` property.

Now that we have done some basic testing of the service and its responses to queries that we want to run, we can build a client to make those queries and process the JSON responses.

Using an ASP.NET Core MVC project as a GraphQL client

We will create a model class to make it easy to deserialize the response:

1. Use your preferred code editor to add a new project, as defined in the following list:
 - Project template: **ASP.NET Core Web App (Model-View-Controller)** / `mvc`
 - Solution file and folder: `ModernServices`
 - Project file and folder:
`Northwind.GraphQL.Client.Mvc`
 - Other Visual Studio options:
 - **Authentication Type:** None
 - **Configure for HTTPS:** Selected
 - **Enable Docker:** Cleared
 - **Do not use top-level statements:** Cleared
10. In the `Northwind.GraphQL.Client.Mvc` project, add a project reference to the Northwind entity models project, as shown in the following markup:

```
<ItemGroup> <ProjectReference Include= "..  
  \..\ModernApps\Northwind.EntityModels  
  \Northwind.EntityModels.csproj" /> </  
ItemGroup>
```

The path to the project must not have a line break.

1. Build the `Northwind.GraphQL.Client.Mvc` project at the command prompt or terminal.
2. In the `Properties` folder, in `launchSettings.json`, modify the `applicationUrl` to use port 5133 for https and port 5134 for http, as highlighted in the following configuration:

```
"https" :{ "commandName": "Project",
"dotnetRunMessages": true,
"launchBrowser": true, "applicationUrl"
"https://localhost:5133;http://
localhost:5134":, "environmentVariables":
{ "ASPNETCORE_ENVIRONMENT": "Development"
}
```

3. In the `Northwind.GraphQL.Client.Mvc` project, in the `Models` folder, add a new class file named `ResponseErrors.cs`, as shown in the following code:

```
namespace
Northwind.GraphQL.Client.Mvc.Models;
public class ResponseErrors { public
Error[]? Errors { get; set; } } public
class Error { public string Message { get;
set; } = null!; public Location[]
Locations { get; set; } = null!; public
string[] Path { get; set; } = null!; }
public class Location { public int Line {
get; set; } public int Column { get; set;
} }
```

4. In the `Models` folder, add a new class file named `ResponseProducts.cs`, as shown in the following code:


```
using Northwind.EntityModels; // To use
Product. namespace
Northwind.GraphQL.Client.Mvc.Models;
public class ResponseProducts { public
class DataProducts { public Product[]?
ProductsInCategory { get; set; } } public
DataProducts? Data { get; set; } }
```

5. In the Models folder, add a new class file named ResponseCategories.cs, as shown in the following code:

```
using Northwind.EntityModels; // To use
Category. namespace
Northwind.GraphQL.Client.Mvc.Models;
public class ResponseCategories { public
class DataCategories { public Category[]?
Categories { get; set; } } public
DataCategories? Data { get; set; } }
```

6. In the Models folder, add a new class file named IndexViewModel.cs, which will have properties to store all the data that we might want to show in the view, as shown in the following code:

```
using Northwind.EntityModels; // To use
Product. using System.Net; // To use
HttpStatusCode. namespace
Northwind.GraphQL.Client.Mvc.Models;
public class IndexViewModel { public
HttpStatusCode Code { get; set; } public
string? RawResponseBody { get; set; }
public Product[]? Products { get; set; }
public Category[]? Categories { get; set; }
} public Error[]? Errors { get; set; } }
```

7. In `Program.cs`, import the namespace to set HTTP headers, as shown in the following code:

```
using System.Net.Http.Headers; // To use
MediaTypesWithQualityHeaderValue.
```

8. In `Program.cs`, after the `CreateBuilder` method call, add statements to register an HTTP client for the GraphQL service, as shown in the following code:

```
builder.Services.AddHttpClient(name:
"Northwind.GraphQL.Service",
configureClient: options => {
options.BaseAddress = new Uri("https://
localhost:5131/");
options.DefaultRequestHeaders.Accept.Add(
new MediaTypeWithQualityHeaderValue(
"application/json", 1.0)); });
```

9. In the `Controllers` folder, in `HomeController.cs`, import the namespace to work with text encodings and for the local project models, as shown in the following code:

```
using Northwind.Mvc.GraphQLClient.Models;
// To use IndexViewModel. using
System.Text; // To use Encoding.
```

10. Define a field to store the registered HTTP client factory and a logger for the controller, and set them in the constructor, as shown in the following code:

```
private readonly ILogger<HomeController>
_logger; protected readonly
IHttpClientFactory _clientFactory; public
HomeController(ILogger<HomeController>
logger, IHttpClientFactory clientFactory)
{ _logger = logger; _clientFactory =
clientFactory; }
```

11. In the `Index` action method, modify the method to be asynchronous. Then, add statements to call the GraphQL service, and note that:

- The HTTP request is a `POST`.
- The media type is for an `application/json` document that contains a GraphQL query.
- The query requests the ID, name, and number of units in stock for all products in a given category, passed as a parameter named `id`, as shown in the following code:

```
public async
Task<IActionResult>
Index(string id = "1") {
    IndexViewModel model = new();
    try { HttpClient client =
        _clientFactory.CreateClient(
            name:
                "Northwind.GraphQL.Service");
        // First, try a simple GET
        request to service root.
        HttpRequestMessage request =
            new( method: HttpMethod.Get,
                requestUri: "/");
        HttpResponseMessage response =
            await
            client.SendAsync(request); if
```

```

(!response.IsSuccessStatusCode)
{
    model.Code =
        response.StatusCode;
    model.Errors = new[] { new
        Error { Message = "Service is
        not successfully responding to
        GET requests." } }; return
        View(model); } // Next, make a
        request to the GraphQL
        endpoint. request = new(
        method: HttpMethod.Post,
        requestUri: "graphql");
    request.Content = new
        StringContent(content: $$$"" {
        "query":
        "{productsInCategory(categoryId:
        {{{id}}}){productId productName
        unitsInStock}}" } "",
        encoding: Encoding.UTF8,
        mediaType: "application/json");
    response = await
        client.SendAsync(request);
    model.Code =
        response.StatusCode;
    model.RawResponseBody = await
        response.Content.ReadAsStringAsync();
    if
        (response.IsSuccessStatusCode)
        {
            model.Products = (await
                response.Content
                .ReadFromJsonAsync<ResponseProducts>())
        }
        else {
            model.Errors = (await
                response.Content
                .ReadFromJsonAsync<ResponseErrors>())?
        }
    }
    catch (Exception ex) {

```

```
_logger.LogWarning(  
    $"Northwind.GraphQL.Service  
    exception: {ex.Message}");  
model.Errors = new[] { new  
    Error { Message = ex.Message }  
}; } return View(model); }
```

Good practice: To set the content of our request, we will use the C# 11 or later raw interpolated string literal syntax of three-dollar signs and three double quotes. This allows us to embed the `id` variable using three curly braces, which should not be confused with the two curly braces after `unitsInStock`, which end the query itself.

1. In the `Views/Home` folder, in `Index.cshtml`, delete its existing markup, and then add markup to render the seafood products, as shown in the following markup:

To save you entering a lot of code, the full file is available at the following link:

[https://github.com/markjprice/
apps-services-net10/blob/main/
code/ModernServices/
Northwind.GraphQL.Client.Mvc/
Views/Home/Index.cshtml](https://github.com/markjprice/apps-services-net10/blob/main/code/ModernServices/Northwind.GraphQL.Client.Mvc/Views/Home/Index.cshtml).

```
@using Northwind.EntityModels @using  
Northwind.GraphQL.Client.Mvc.Models @* for VS
```

```

Code only *@ @model IndexViewModel @{
ViewData["Title"] = "Products from GraphQL
service"; } <div class="text-center"> <h1
class="display-4">@ViewData["Title"]</h1> <div
class="card card-body"> <form> Enter a
category id <input name="id" value="1" />
<input type="submit" /> </form> </div> @if
(Model.Errors is not null) { <div class="alert
alert-danger" role="alert"> <table
class="table table-striped"> <thead> <tr>
<td>Message</td> <td>Path</td> <td>Locations</
td> </tr> </thead> <tbody> @foreach (Error
error in Model.Errors) { <tr>
<td>@error.Message</td> <td> @if (error.Path
is not null) { @foreach (string path in
error.Path) { <span class="badge bg-
danger">@path</span> } } </td> <td> @if
(error.Locations is not null) { @foreach
(Location location in error.Locations) { <span
class="badge bg-danger"> @location.Line,
@location.Column </span> } } </td> </tr> } </
tbody> </table> </div> } @if (Model.Categories
is not null) { <div> <p class="alert alert-
success" role="alert"> There are
@Model.Categories.Count() products.</p> <p>
@foreach (Category category in
Model.Categories) { <span class="badge bg-
dark"> @category.CategoryId
@category.CategoryName </span> } </p> </div> }
@if (Model.Products is not null) { <div> <p
class="alert alert-success" role="alert">
There are @Model.Products.Count() products.</
p> <p> @foreach (Product p in Model.Products)
{ <span class="badge bg-dark"> @p.ProductId
@p.ProductName - @(p.UnitsInStock is null ?

```

```
"0" : p.UnitsInStock.Value) in stock </span> }
</p> </div> } <p> <a class="btn btn-primary"
data-bs-toggle="collapse"
href="#collapseExample" role="button" aria-
expanded="false" aria-
controls="collapseExample"> Show/Hide Details
</a> </p> <div class="collapse"
id="collapseExample"> <div class="card card-
body"> Status code @( (int)Model.Code):
@Model.Code <hr /> @Model.RawResponseBody </
div> </div> </div>
```

Testing the .NET client

Now, we can test our .NET client:

1. If your database server is not running, for example, because you are hosting it in Docker, a virtual machine, or in the cloud, then make sure to start it.
2. Start the `Northwind.GraphQL.Service` project, using its `https` profile without debugging.
3. Start the `Northwind.GraphQL.Client.Mvc` project, using its `https` profile without debugging.
4. Note that products are successfully retrieved using GraphQL, as shown in *Figure 13.9*:

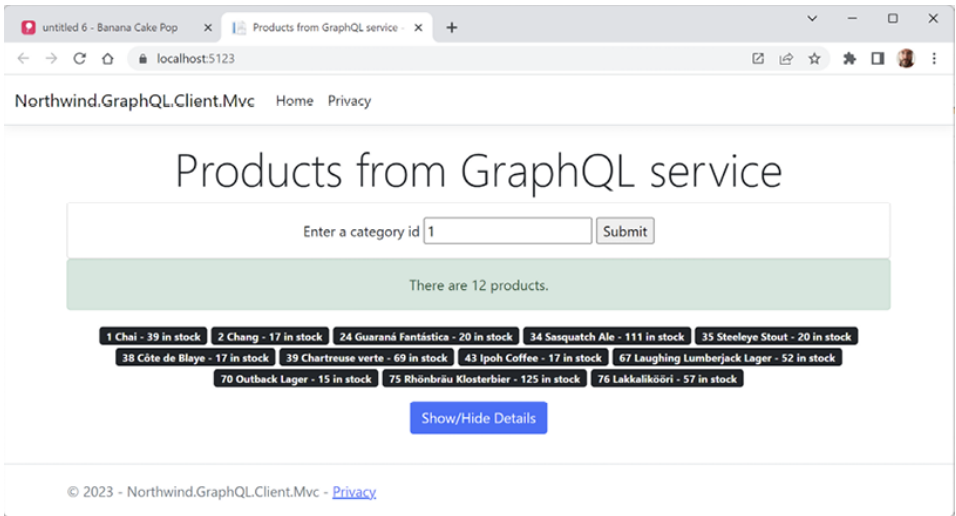


Figure 13.9: Products in the Beverages category from the GraphQL service

1. Enter another category ID that exists, for example, 4.
2. Enter a category ID that is out of range, for example, 13, and note that 0 products are returned.
3. Close Chrome, and shut down the web server for the `Northwind.GraphQL.Client.Mvc` project.
4. In `HomeController.cs`, modify the query to make a deliberate mistake, like changing `productId` to `productid`.
5. Start the `Northwind.GraphQL.Client.Mvc` project, using the `https` profile without debugging.
6. Click the **Show/Hide Details** button, and note the error message and response details, as shown in the following output:

```
{
  "errors": [
    {
      "message": "The field \u0060productid\u0060 does not exist on the type \u0060Product\u0060.",
      "locations": [
        {
          "line": 1,
          "column": 35
        }
      ],
      "path": [
        "productsInCategory"
      ],
      "extensions": {}
    }
  ]
}
```



```
{"type": "Product", "field": "productid",  
  "responseName": "productid",  
  "specifiedBy": "http://spec.graphql.org/  
October2021/ #sec-Field-Selections-on-  
Objects-Interfaces-and-Unions-Types" } } }
```

7. Close Chrome, and shut down both web servers.
8. Fix the mistake in the query!

You've now seen that any client that can make an HTTP request and process HTTP responses can send queries to our GraphQL service and get back JSON that can be easily processed as one would expect. We used Hot Chocolate to build our service. To make it even easier to build clients, we can use another ChilliCream product called Strawberry Shake.

Creating a console app client using Strawberry Shake

Instead of using ordinary HTTP clients, ChilliCream has a GraphQL client library to more easily build .NET clients for GraphQL services.

You can learn more about Strawberry Shake at the following link: <https://chillicream.com/docs/strawberryshake>.

Now, let's create another client using Strawberry Shake so that you can see the benefits:

1. Use your preferred code editor to add a new **Console App** / `console` project, named `Northwind.GraphQL.Client.Console`.
2. At the command prompt or terminal for the project folder, create a tool manifest file, as shown in the following command:

```
dotnet new tool-manifest
```

3. At the command line or terminal, install the Strawberry Shake tools, as shown in the following command:

```
dotnet tool install StrawberryShake.Tools  
--local
```

Be patient as installation can take a few minutes and no progress indication is shown.

1. Note Strawberry Shake is installed, as shown in the following output:

```
You can invoke the tool from this  
directory using the following commands:  
'dotnet tool run dotnet-graphql' or  
'dotnet dotnet-graphql'. Tool  
'strawberryshake.tools' (version  
'15.1.11') was successfully installed.  
Entry is added to the manifest file C:  
\apps-services-net10\ModernServices\  
Northwind.GraphQL.Client.Console\.config  
\dotnet-tools.json.
```

2. In the project, treat warnings as errors, add references to NuGet packages for Microsoft extensions for dependency injection, working with HTTP, and Strawberry Shake code generation, and then globally and statically import the `Console` class, as highlighted in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk">
```

```

<PropertyGroup> <OutputType>Exe</
OutputType> <TargetFramework>net10.0</
TargetFramework> <ImplicitUsings>enable</
ImplicitUsings> <Nullable>enable</
Nullable> <TreatWarningsAsErrors>true</
TreatWarningsAsErrors> </PropertyGroup>
<ItemGroup> <PackageReference
Include="Microsoft.Extensions.DependencyInjection"
/> <PackageReference
Include="Microsoft.Extensions.Http" />
<PackageReference
Include="StrawberryShake.Server" /> </
ItemGroup> <ItemGroup> <Using
Include="System.Console" Static="true" />
</ItemGroup> </Project>

```

You need to use different Strawberry Shake packages for different types of .NET project. For console apps and ASP.NET Core apps, reference `StrawberryShake.Server`. For Blazor WebAssembly apps, reference `StrawberryShake.Blazor`. For .NET MAUI apps, reference `StrawberryShake.Maui`.

1. Build the `Northwind.GraphQL.Client.Console` project to restore packages.
2. Start the `Northwind.GraphQL.Service` project, using the `https` profile without debugging, and leave it running so that the Strawberry Shake tool can talk to it.
3. In the `Northwind.GraphQL.Client.Console` project, at the command prompt or terminal, add a client for your GraphQL service, as shown in the following command:

```
dotnet graphql init https://  
localhost:5131/graphql/ -n NorthwindClient
```

4. Note the results, as shown in the following output:

```
Download schema started. Download schema  
completed in 189 ms Client configuration  
started. Client configuration completed in  
83 ms
```

5. In the `Northwind.GraphQL.Client.Console` project, in the `.graphqlrc.json` file, add an entry to control the C# namespace used during code generation, as highlighted in the following markup:

```
{ "schema": "schema.graphql", "documents":  
  "**/*.graphql", "extensions": {  
    "strawberryShake": { "name":  
      "NorthwindClient", "namespace"  
        "Northwind.GraphQL.Client.Console":,  
      "url": "https://localhost:5131/graphql/",  
      "records": { "inputs": false, "entities":  
        false }, "transportProfiles": [ {  
        "default": "Http", "subscription":  
        "WebSocket" } ] } } }
```

6. In the `Northwind.GraphQL.Client.Console` project, add a new file named `seafoodProducts.graphql`, which defines a query to get seafood products, as shown in the following document:

```
query SeafoodProducts {  
  productsInCategory(categoryId:8) {
```

```
productId productName unitsInStock } }
```

GraphQL queries used by Strawberry Shake must be named.

1. If you are using Visual Studio, it might automatically modify the project file to explicitly remove this file from the build process because it does not recognize it. If it has, then delete or comment out that element, as shown in the following markup:

```
<!-- An element like this will remove the  
file from the build process. <ItemGroup>  
<GraphQL Remove="seafoodProducts.graphql"  
</ItemGroup> -->
```

Good practice: There must be at least one .graphql file for the Strawberry Shake tool to be able to generate its code automatically. An element like the preceding one will prevent the Strawberry Shake tool from generating its code, and you will later get compile errors. You should delete or comment out that element.

1. Build the `Northwind.GraphQL.Client.Console` project to make Strawberry Shake process the GraphQL query file and generate proxy classes.
2. Note the `obj\Debug\net10.0\berry` folder that was autogenerated, the file named `NorthwindClient.Client.cs`, and the dozen or so types defined by it, including the `INorthwindClient` interface, as shown in *Figure 13.10*:

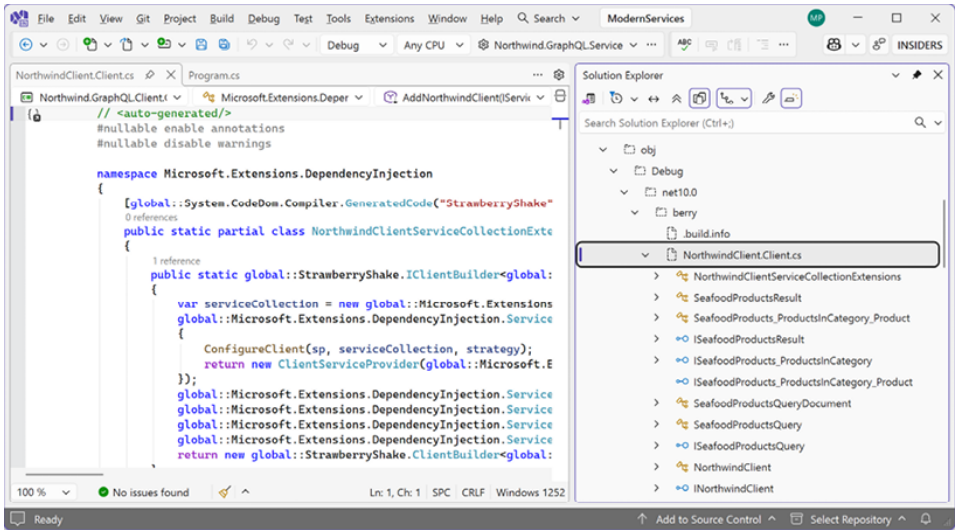


Figure 13.10: The generated class file for the Northwind GraphQL service

1. In `Program.cs`, delete the existing statements. Add statements to create a new service collection, add the autogenerated `NorthwindClient` to it with the correct URL for the service, and then get and use the dependency service to fetch the seafood products, as shown in the following code:

```
using
Microsoft.Extensions.DependencyInjection;
// To use ServiceCollection. using
Northwind.GraphQL.Client.Console; // To
use INorthwindClient. using
StrawberryShake; // To use EnsureNoErrors
extension method. ServiceCollection
serviceCollection = new();
serviceCollection .AddNorthwindClient() //
Strawberry Shake extension method.
.ConfigureHttpClient(client =>
client.BaseAddress = new Uri("https://
localhost:5131/graphql"));
```

```

IServiceProvider services =
serviceCollection.BuildServiceProvider();
INorthwindClient client =
services.GetRequiredService<INorthwindClient>();
var result = await
client.SeafoodProducts.ExecuteAsync();
result.EnsureNoErrors(); if (result.Data
is null) { WriteLine("No data!"); return;
} foreach (var product in
result.Data.ProductsInCategory) {
WriteLine("{0}: {1}", product.ProductId,
product.ProductName); }

```

2. Run the console app and note the results, as shown in the following output:

```

10: Ikura 13: Konbu 18: Carnarvon Tigers
30: Nord-Ost Matjeshering 36: Inlagd Sill
37: Gravad lax 40: Boston Crab Meat 41:
Jack's New England Clam Chowder 45: Rogede
sild 46: Spegesild 58: Escargots de
Bourgogne 73: Röd Kaviar

```

When I introduced you to GraphQL, I mentioned that GraphQL is not just about queries. It also has capabilities to modify data and subscribe to data. Let's look at those features now.

Implementing GraphQL mutations

Most services need to modify data as well as query it. GraphQL calls these **mutations**. A mutation has three related components:

- The mutation itself, which defines the change that will be made to the

graph. It should be named using a verb, a noun, and use camel casing, for example, `addProduct`.

- The **input** is the input for a mutation, and it should have the same name as the mutation with a suffix of `Input`, for example, `AddProductInput`. Although there is only one input, it is an object graph, so it can be as complex as you need.
- The **payload** is the returned document for a mutation, and it should have the same name as the mutation with a suffix of `Payload`, for example, `AddProductPayload`. Although there is only one payload, it is an object graph, so it can be as complex as you need.

Adding mutations to the GraphQL service

Let's define mutations for adding, and later, we will define some to update and delete products:

1. In the `Northwind.GraphQL.Service` project/folder, add a class file named `Mutation.cs`.
2. In the class file, define a record and two classes to represent the three types needed to perform an `addProduct` mutation using standard EF Core code that you should already be familiar with, as shown in the following code:

```
using Northwind.EntityModels; // To use
Product. namespace
Northwind.GraphQL.Service; // Inputs are
readonly so we will use a record. public
record AddProductInput( string
ProductName, int? SupplierId, int?
CategoryId, string QuantityPerUnit,
decimal? UnitPrice, short? UnitsInStock,
short? UnitsOnOrder, short? ReorderLevel,
bool Discontinued); public class
AddProductPayload { public
```



```

AddProductPayload(Product product) {
    Product = product; } public Product
Product { get; } } public class Mutation {
public async Task<AddProductPayload>
AddProductAsync( AddProductInput input,
NorthwindContext db) { // This could be a
good place to use a tool like AutoMapper,
// but we will do the mapping between two
objects manually. Product product = new()
{ ProductName = input.ProductName,
SupplierId = input.SupplierId, CategoryId
= input.CategoryId, QuantityPerUnit =
input.QuantityPerUnit, UnitPrice =
input.UnitPrice, UnitsInStock =
input.UnitsInStock, UnitsOnOrder =
input.UnitsOnOrder, ReorderLevel =
input.ReorderLevel, Discontinued =
input.Discontinued };
db.Products.Add(product); int affectedRows
= await db.SaveChangesAsync(); // We could
use affectedRows to return an error // or
some other action if it is 0. return new
AddProductPayload(product); } }

```

3. In Program.cs, add a call to the AddMutationType<T> method to register your Mutation class, as highlighted in the following code:

```

builder.Services .AddGraphQLServer()
.AddFiltering() .AddSorting()
.RegisterDbContext<NorthwindContext>()
.AddQueryType<Query>()
.AddMutationType<Mutation>();

```

Exploring the add product mutation

Now, we can explore mutations using Nitro:

1. Start the `Northwind.GraphQL.Service` project, using the `https` profile without debugging.
2. In **Nitro**, click the **+** to open a new tab.
3. Click the **Schema** tab, click the **SDL** tab, and note the mutation type, as shown in the following code:

```
type Mutation { addProduct(input:
AddProductInput!): AddProductPayload! }
type Product { productId: Int!
productName: String! supplierId: Int
categoryId: Int quantityPerUnit: String
unitPrice: Decimal unitsInStock: Short
unitsOnOrder: Short reorderLevel: Short
discontinued: Boolean! category: Category
supplier: Supplier orderDetails:
[OrderDetail!]! } ... type
AddProductPayload { product: Product! }
input AddProductInput { productName:
String! supplierId: Int categoryId: Int
quantityPerUnit: String! unitPrice:
Decimal unitsInStock: Short unitsOnOrder:
Short reorderLevel: Short discontinued:
Boolean! }
```

4. Click the **Operation** tab and, if necessary, create a new blank document. Then, in the **Request** section, enter a mutation to add a new product named `Tasty Burgers`. After that, from the returned `product` object, just select the `ID` and `name`, as shown in the following code:

```
mutation AddProduct { addProduct( input: {
```

```
productName: "Tasty Burgers" supplierId: 1
categoryId: 2 quantityPerUnit: "6 per box"
unitPrice: 40 unitsInStock: 0
unitsOnOrder: 0 reorderLevel: 0
discontinued: false } ) { product {
productId productName } } }
```

5. Click **Run**, and note that the new product has been successfully added and assigned the next sequential number by the SQL Server database, which could be any number over 77, depending on if you have already added some other products, as shown in the following output and in *Figure 13.11*:

```
{ "data": { "addProduct": { "product": {
"productId": 78, "productName": "Tasty
Burgers", } } } }
```

Warning! Please make a note of the ID assigned to the new product you have added. In the next section, you will update this product and then delete it. You cannot delete any of the existing products with IDs between 1 and 77 because they are related to other tables, and doing so would throw a referential integrity exception!

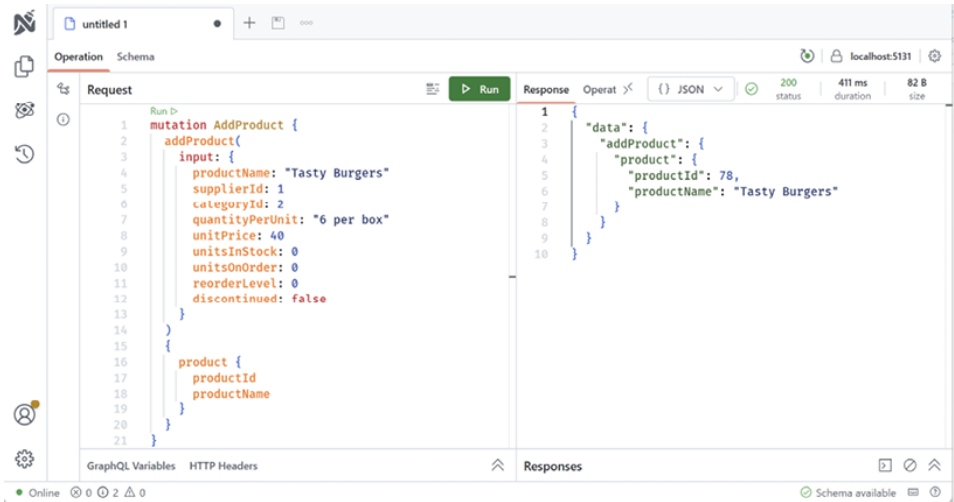


Figure 13.11: Adding a new product using a GraphQL mutation

1. Close the browser, and shut down the web server.

Implementing updates and deletes as mutations

Next, we will define mutations to update just the unit price for a product, as well as all the “units” fields for a product, plus delete a product:

1. In `Mutation.cs`, define three record types to represent the inputs needed to perform two `updateProduct` and one `deleteProduct` mutations, as shown in the following code:

```

public record UpdateProductPriceInput (
    int? ProductId, decimal? UnitPrice);
public record UpdateProductUnitsInput (
    int? ProductId, short? UnitsInStock,
    short? UnitsOnOrder, short? ReorderLevel);
public record DeleteProductInput ( int?
    ProductId);

```

2. In `Mutation.cs`, define two class types to represent the types needed to return the results from an `update` or `delete` mutation, including if the action was successful, as shown in the following code:

```
public class UpdateProductPayload { public
UpdateProductPayload(Product? product,
bool updated) { Product = product; Success
= updated; } public Product? Product {
get; } public bool Success { get; } }
public class DeleteProductPayload { public
DeleteProductPayload(bool deleted) {
Success = deleted; } public bool Success {
get; } }
```

3. In `Mutation.cs`, in the `Mutation` class, define three methods to implement two `updateProduct` and one `deleteProduct` mutations using standard EF Core code that you should already be familiar with, as shown in the following code:

```
public async Task<UpdateProductPayload>
UpdateProductPriceAsync(
UpdateProductPriceInput input,
NorthwindContext db) { Product? product =
await
db.Products.FindAsync(input.ProductId);
int affectedRows = 0; if (product is not
null) { product.UnitPrice =
input.UnitPrice; affectedRows = await
db.SaveChangesAsync(); } return new
UpdateProductPayload(product, updated:
affectedRows == 1); } public async
Task<UpdateProductPayload>
UpdateProductUnitsAsync(
```

```

UpdateProductUnitsInput input,
NorthwindContext db) { Product? product =
await
db.Products.FindAsync(input.ProductId);
int affectedRows = 0; if (product is not
null) { product.UnitsInStock =
input.UnitsInStock; product.UnitsOnOrder =
input.UnitsOnOrder; product.ReorderLevel =
input.ReorderLevel; affectedRows = await
db.SaveChangesAsync(); } return new
UpdateProductPayload(product, updated:
affectedRows == 1); } public async
Task<DeleteProductPayload>
DeleteProductAsync( DeleteProductInput
input, NorthwindContext db) { Product?
product = await
db.Products.FindAsync(input.ProductId);
int affectedRows = 0; if (product is not
null) { db.Products.Remove(product);
affectedRows = await
db.SaveChangesAsync(); } return new
DeleteProductPayload( deleted:
affectedRows == 1); }

```

4. If your database server is not running, for example, because you are hosting it in Docker, a virtual machine, or in the cloud, then make sure to start it.
5. Start the `Northwind.GraphQL.Service` project, using the `https` profile without debugging.
6. In **Nitro**, click the **+** to open a new tab.
7. Write a named query to request products that you have added, for example, with a `productId` greater than `77`, as shown in the following markup:

```
query NewProducts { products(where: {  
  productId: { gt: 77 } }) { productId  
  productName unitPrice unitsInStock  
  unitsOnOrder reorderLevel } }
```

8. Click **Run**, and note the response includes the new product you previously added with a unit price of 40, as shown in the following output:

```
{ "data": { "products": [ { "productId":  
78, "productName": "Tasty Burgers",  
"unitPrice": 40, "unitsInStock": 0,  
"unitsOnOrder": 0, "reorderLevel": 0 } ] }  
}
```

9. Make a note of the `productId` of the product you added. In my case, it is 78.
10. In **Nitro**, click the **+** to open a new tab.
11. Enter a mutation to update the unit price of your new product to 75, and then from the returned `product` object, just select the ID, name, unit price, and units in stock, as shown in the following code:

```
mutation UpdateProductPrice {  
  updateProductPrice( input: { productId: 78  
    unitPrice: 75 } ) { product { productId  
    productName unitPrice unitsInStock } } }
```

12. Click **Run**, and note the response, as shown in the following output:

```
{ "data": { "updateProductPrice": {  
  "product": { "productId": 78,
```

```
"productName": "Tasty Burgers",  
"unitPrice": 75, "unitsInStock": 0 } } } }
```

13. Click the **+** to open a new tab, enter a mutation to update the units of an existing product, and then from the returned `product` object just select the `success` property, as shown in the following code:

```
mutation UpdateProductUnits {  
  updateProductUnits( input: { productId: 78  
    unitsInStock: 20 unitsOnOrder: 0  
    reorderLevel: 10 } ) { success } }
```

14. Click **Run**, and note the response, as shown in the following output:

```
{ "data": { "updateProductUnits": {  
  "success": true } } }
```

15. In the query tab, to request new products (`products (where: { productId: { gt: 77 } })`), click **Run**, and note the response, as shown in the following output:

```
{ "data": { "products": [ { "productId":  
  78, "productName": "Tasty Burgers",  
  "unitPrice": 75, "unitsInStock": 20,  
  "unitsOnOrder": 0, "reorderLevel": 10 } ]  
} }
```

16. Click the **+** to open a new tab, and enter a mutation to delete the product and show if it was successful, as shown in the following code:

```
mutation DeleteProduct { deleteProduct(
```



```
input: { productId: 78 } ) { success } }
```

Warning! You will not be able to delete products that are referenced in other tables. IDs 1 to 77 will throw a referential integrity exception.

1. Click **Run**, and note the response, as shown in the following output:

```
{ "data": { "deleteProduct": { "success": true } } }
```

2. Confirm that the product was deleted by re-running the query for new products, and note that you get an empty array, as shown in the following output:

```
{ "data": { "products": [] } }
```

3. Close the browser, and shut down the web server.

Now you know how to add, update, and delete data using GraphQL mutations. Let's end this chapter by looking at what you can do with GraphQL subscriptions.

Implementing GraphQL subscriptions

GraphQL subscriptions, by default, work over WebSockets but can also work over **Server-Sent Events (SSE)**, SignalR, or even gRPC.

Imagine that a client app wants to be notified when a product has its unit price reduced. It would be great if the client could subscribe to an event that gets triggered whenever a unit price is reduced, instead of having to query for

changes to the unit prices.

Adding a subscription and topic to the GraphQL service

Based on the example just mentioned, let's add this feature to our GraphQL service using subscriptions:

1. Add a new class file named `ProductDiscount.cs`.
2. Modify the contents to define a model to notify a client about a product's unit price reduction, as shown in the following code:

```
namespace Northwind.GraphQL.Service;
public class ProductDiscount { public int?
ProductId { get; set; } public decimal?
OriginalUnitPrice { get; set; } public
decimal? NewUnitPrice { get; set; } }
```

3. Add a new class file named `Subscription.cs`.
4. Modify the contents to define a subscription to an event (aka a topic) named `OnProductDiscounted` that a client can subscribe to, as shown in the following code:

```
namespace Northwind.GraphQL.Service;
public class Subscription { [Subscribe]
[Topic] public ProductDiscount
OnProductDiscounted( [EventMessage]
ProductDiscount productDiscount) =>
productDiscount; }
```

5. In `Mutation.cs`, in the `UpdateProductPriceAsync` method, add statements to send a message over the topic whenever a product has

its unit price reduced, as highlighted in the following code:

```
public async Task<UpdateProductPayload>
UpdateProductPriceAsync(
    UpdateProductPriceInput input,
    NorthwindContext db, ITopicEventSender
eventSender) { Product? product = await
    db.Products.FindAsync(input.ProductId);
    int affectedRows = 0; if (product is not
    null) { if (input.UnitPrice <
    product.UnitPrice) { // If the product has
    been discounted, // send a message to
    subscribers. ProductDiscount
    productDiscount = new() { ProductId =
    input.ProductId, OriginalUnitPrice =
    product.UnitPrice, NewUnitPrice =
    input.UnitPrice }; await
    eventSender.SendAsync(topicName:
    nameof(Subscription.OnProductDiscounted),
    message: productDiscount); }
    product.UnitPrice = input.UnitPrice;
    affectedRows = await
    db.SaveChangesAsync(); } return new
    UpdateProductPayload(product, updated:
    affectedRows == 1); }
```

6. In Program.cs, configure the GraphQL service to register the Subscription class and to store active subscriptions in-memory, as highlighted in the following code:

```
builder.Services .AddGraphQLServer()
    .AddFiltering() .AddSorting()
    .AddSubscriptionType<Subscription>()
    .AddInMemorySubscriptions()
```

```
.RegisterDbContext<NorthwindContext>()  
.AddQueryType<Query>()  
.AddMutationType<Mutation>();
```

As well as in-memory, you can use Redis and other data stores to keep track of active subscriptions.

1. Optionally, after building the app, configure the use of WebSockets, as shown in the following code:

```
app.UseWebSockets(); // For subscriptions.
```

This is optional because the GraphQL service can fall back to using SSE over https.

Exploring subscribing to a topic

Let's subscribe to a topic and see the results:

1. Start the `Northwind.GraphQL.Service` project, using the https profile without debugging.
2. In **Nitro**, click the **+** to open a new tab.
3. Click the **Schema** tab, click the **Subscription** type, and note the topic named **onProductDiscounted**, as shown in *Figure 13.12*:

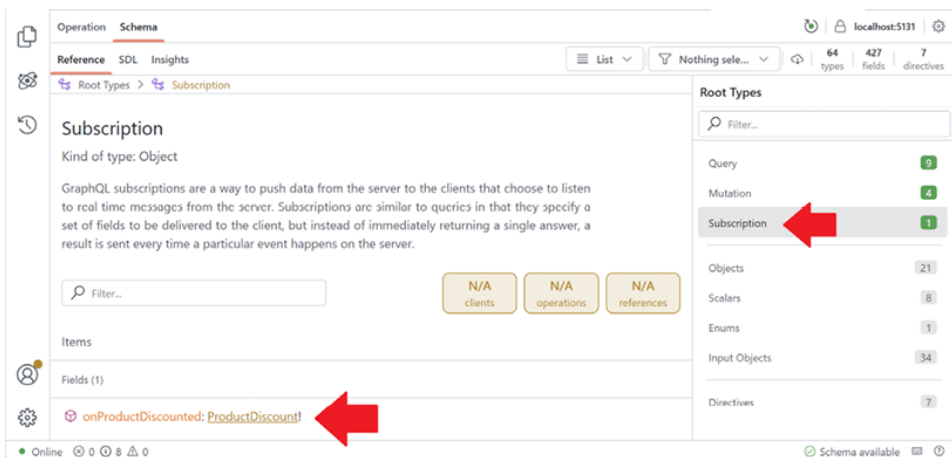


Figure 13.12: A subscription with a topic

1. Click **Operation**, enter a subscription to the topic, and choose all the fields to show in the results, as shown in the following code:

```
subscription { onProductDiscounted {  
  productId originalUnitPrice newUnitPrice }  
}
```

2. Click **Run**, and note that the subscription starts, but no results are shown yet.
3. Click the + to open a new tab, and enter a mutation to update the unit price of the existing product 1 to 8.99. Then, from the returned product object, select the ID and unit price, and show if the update succeeded, as shown in the following code:

```
mutation UpdateProductPrice {  
  updateProductPrice( input: { productId: 1  
    unitPrice: 8.99 } ) { product { productId  
    unitPrice } success } }
```

4. Click **Run**, and note the response, as shown in the following output:

```
{ "data": { "updateProductPrice": {  
  "product": { "productId": 1, "unitPrice":  
    8.99 }, "success": true } } }
```

5. Switch back to the tab with the subscription, and note the response and that the subscription is still active, indicated by the spinners on the tab and the **Cancel** button, as shown in *Figure 13.13*:

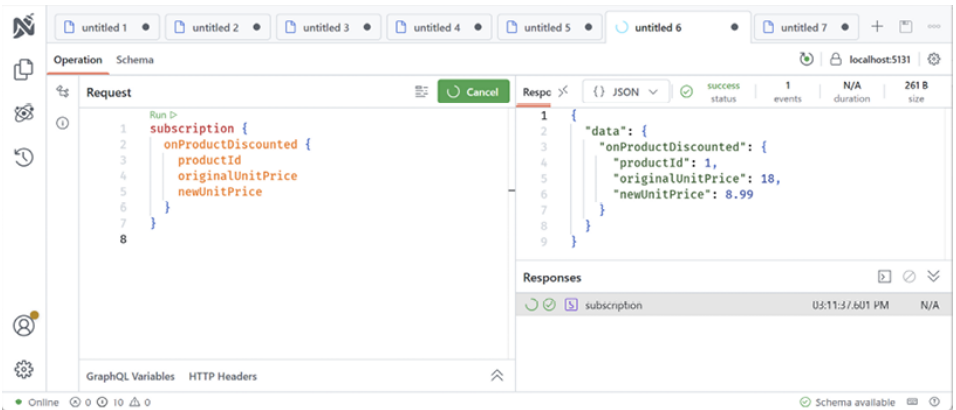


Figure 13.13: An active subscription shows a spinner

1. Switch back to the tab with the update mutation, change the unit price to 7.99, and click **Run**. Then, switch back to the tab with the subscription, and note that it receives that update notification too.
2. Switch back to the tab with the update mutation, change the unit price to 9.99, and click **Run**. Then, switch back to the tab with the subscription, and note that it has not been sent any notifications because the unit price was increased, not reduced.
3. Close the browser, and shut down the web server.

GraphQL subscription summary

To summarize GraphQL subscriptions, they are:

- A typed, schema-defined event stream
- Usually transported over WebSockets but also supports other transports like Server-Sent Events
- Expressed using the same query language as GraphQL queries and mutations

The client says, “Whenever this event happens, send me data shaped like this.” That shape is enforced by the GraphQL schema.

The benefits of GraphQL subscriptions over implementing your own service using SignalR are:

- **Strongly defined data shape at the protocol level.** With GraphQL subscriptions, the schema defines exactly what can be sent, clients explicitly select fields, and you cannot accidentally send extra junk. This is excellent for large front-end teams, public APIs, and multiple consumers with different data needs. SignalR has zero control here. You can send anything, break anything, and only find out at runtime.
- **Single mental model for reads, writes, and live updates.** GraphQL gives you queries for reads, mutations for writes, and subscriptions for live updates. All of them use the same schema, use the same types, and use the same tooling. That consistency matters if your entire API surface is GraphQL and your frontend team already speaks GraphQL fluently.
- **Client-driven payload selection.** With subscriptions, the client controls which fields it wants and how deeply nested the data is. Two clients can subscribe to the same event and receive different shapes of data without the server caring. With SignalR the server pushes one payload, all clients get the same thing, and customization means branching logic or multiple hub methods.

But you should keep in mind that GraphQL subscriptions are not perfect in all scenarios, for example:

- **SignalR is vastly more flexible.** SignalR can do things GraphQL

subscriptions simply cannot, like arbitrary server-to-client commands, client-to-client messaging, presence tracking, typing indicators, group management, and connection lifecycle events. GraphQL subscriptions are data streams, not conversations. If your problem smells like chat, collaboration, dashboards, or multiplayer anything, SignalR fits naturally.

- **GraphQL subscriptions are not event buses.** A common mistake is thinking, “We’ll use GraphQL subscriptions for all our events.” That usually ends badly. Subscriptions work best for UI-driven updates, small, well-defined event streams, and data that mirrors queries. SignalR works for anything real-time that doesn’t look like a query result.

Practicing and exploring

Test your knowledge and understanding by answering some questions, getting some hands-on practice, and exploring this chapter’s topics with deeper research.

Exercise 13.1 – Online material

Online material can be extra content written by me for this book, or it can be references to content created by Microsoft or third parties.

Introduction to GraphQL

Learn about GraphQL, how it works, and how to use it: <https://graphql.org/learn/>.

Exercise 13.2 – Practice building .NET clients

In `HomeController.cs`, add an action method named `Categories`, and implement it to query the `categories` field with a variable for the `id`. On the page, allow the visitor to submit an `id`, and note the category

information and a list of its products.

Exercise 13.3 – Test your knowledge

Answer the following questions:

1. What transport protocol does a GraphQL service use?
2. What media type does GraphQL use for its queries?
3. How can you parameterize GraphQL queries?
4. What are the benefits of using Strawberry Shake over a regular HTTP client for GraphQL queries?
5. How might you insert a new product into the Northwind database?

Exercise 13.4 – Explore topics

Use the links on the following page to learn more details about the topics covered in this chapter: <https://github.com/markjprice/apps-services-net10/blob/main/docs/book-links.md#chapter-13---combining-data-sources-using-graphql>.

Summary

In this chapter, you learned about:

- Some of the concepts of GraphQL.
- How to build a `Query` class with fields that represent entities that can be queried.
- How to use the Nitro tool to explore a GraphQL service schema.
- How to use the REST Client extension to POST to a GraphQL service.
- How to create a .NET client for a GraphQL service.
- How to implement GraphQL mutations.
- How to implement GraphQL subscriptions.

In the next chapter, you will learn about the gRPC service technology that can be used to implement efficient microservices.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow this QR code:

https://packt.link/apps_and_services_dotnet10



Join .NETPro – It's Free

Staying sharp in .NET takes more than reading release notes. It requires real-world tips, proven patterns, and scalable solutions. That's what .NETPro, Packt's new newsletter, is all about.

Scan the QR code or visit the link to subscribe:

<https://landing.packtpub.com/dotnetpronewsletter/>



14

Building Efficient Microservices Using gRPC

In this chapter, you will be introduced to gRPC, which enables a developer to build services that can communicate highly efficiently across most platforms.

However, web browsers do not have full support for programmatic access to all features of HTTP/2, which is required by gRPC. This makes gRPC most useful for implementing intermediate tier-to-tier services and microservices because they must perform a lot of communication between multiple microservices to achieve a complete task. Improving the efficiency of that communication is vital to the success of the scalability and performance of microservices.

A modular monolithic, two-tier, client-to-service style service is inherently more efficient because the communication between modules is in-process, and there is only one layer of network communication between the whole service and the clients.

You can learn more about architectural styles in my companion book, *Tools and Skills for .NET 10*.

Microservice architecture has more tiers, and therefore more layers of network communication between the many microservices. It becomes more important to

have highly efficient communication between those layers, and gRPC is designed to be ultra-efficient for network communication, as shown in *Figure 14.1*:

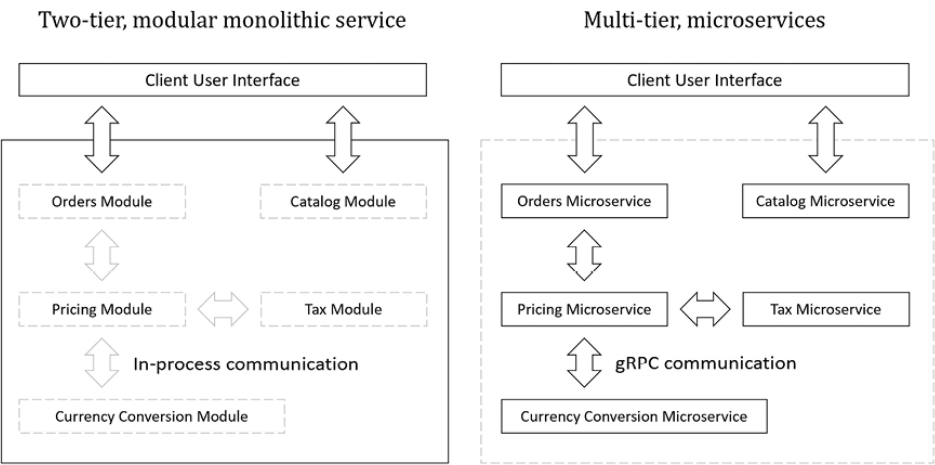


Figure 14.1: Comparing a two-tier modular monolithic service with multi-tier microservices

Good practice: It has been fashionable in the past decade or so to assume that microservices are best for all scenarios, and so for a new system to immediately be implemented using cool microservices rather than as a traditional monolith. More recently, there has been a pushback against this assumption. The industry seems to be settling on the recommendation to start by implementing a system as a modular monolith. Only later, if necessary, should you break the modules apart into actual microservices. As well as being inherently slower due to the extra network communication between microservices, you also need to consider whether the extra coordination and complexity of deployments and orchestration of microservices is

worth it.

This chapter will cover the following topics:

- Understanding gRPC
- Building a gRPC service and client
- Implementing gRPC for an EF Core model
- Taking gRPC further
- Handling dates, times, and decimal numbers
- Implementing interceptors and handling faults
- Implementing gRPC JSON transcoding

Understanding gRPC

gRPC is a modern, open-source, high-performance **Remote Procedure Call (RPC)** framework that can run in any environment. An RPC is when one computer calls a procedure in another process or on another computer over a network as if it were calling a local procedure. It is an example of a client-server architecture.

You can learn more about RPCs at the following link: https://en.wikipedia.org/wiki/Remote_procedure_call.

gRPC was created inside Google as the successor to an internal system called Stubby, which Google had been using for years to connect services at massive scale. When Google open-sourced it in 2015, the name gRPC literally meant: Google Remote Procedure Call. This is confirmed in Google's own early documentation and announcements around the initial open-source release. Over time, Google stopped officially expanding the "g" because the

project gained non-Google governance and adoption, and “Google RPC” sounded corporate and exclusionary. So now you’ll often just see the deliberately vague “gRPC”.

gRPC benefits

gRPC minimizes network usage by using **Protobuf** binary serialization that is not human-readable, unlike JSON or XML used by web services.

gRPC requires HTTP/2, which provides significant performance benefits over earlier versions, like binary framing and compression, and multiplexing of HTTP/2 calls over a single connection.

Binary framing refers to how HTTP messages are transferred between the client and server. HTTP/1.x uses newline delimited plaintext. HTTP/2 splits communication into smaller messages (frames) that are encoded in binary format. Multiplexing means combining multiple messages from different sources into a single message to more efficiently use a shared resource like a network transport.

If you are interested in more details about HTTP/2 and how it makes gRPC more efficient, you can read about it at the following link: <https://grpc.io/blog/grpc-on-http2/>.

gRPC limitations

The main limitation of gRPC is that it cannot be used in web browsers because no browser provides the level of control required to support a gRPC client. For example, browsers do not allow a caller to require that HTTP/2 be used.

Another limitation for developers is that, due to the binary format of the messages, it is harder to diagnose and monitor issues. Many tools do not

understand the format and cannot show messages in a human-readable format.

There is an initiative called **gRPC-Web** that adds an extra proxy layer, and the proxy forwards requests to the gRPC server. However, it only supports a subset of gRPC due to the limitations listed above.

How gRPC works

A gRPC service developer defines a service interface for the methods that can be called remotely, including defining the method parameters and return types. The service implements this interface and runs a gRPC server to handle client calls.

On the client, a strongly typed gRPC client provides the same methods as on the server.

Defining gRPC contracts with .proto files

gRPC uses contract-first API development that supports language-agnostic implementations. A **contract** in this case is an agreement that a service will expose a defined list of methods with specified parameters and return types that implement a prescribed behavior. A client that wishes to call the service can be certain that the service will continue to conform to the contract over time. For example, although new methods might be added, existing ones will never change or be removed.

You write the contracts using `.proto` files that have their own language syntax and then use tools to convert them into various languages like C#. The `.proto` files are used by both the server and client to exchange messages in the correct format.

Here's an example `.proto` file using `proto3` syntax to define a message request that uses a custom `enum`:

```
// Setting the syntax must be first non-  
comment line. syntax = "proto3"; // proto2 is
```



```

the default. /* When this .proto file is used
in a .NET project, it will use the following
C# namespace for the auto-generated code
files. */ option csharp_namespace =
"Northwind.Grpc.Service"; enum SearchType {
SEARCHTYPE_UNSPECIFIED = 0;
SEARCHTYPE_STARTSWITH = 1; SEARCHTYPE_CONTAINS
= 2; SEARCHTYPE_ENDSWITH = 3; } message
SearchRequest { string query = 1; // Fields
must have order numbers. SearchType
search_type = 2; int32 page = 3; int32
page_size = 4; } message SearchResponse { /*
Message types can be nested and/or repeated to
create the equivalent of collections or
arrays. */ repeated SearchResult results = 1;
} message SearchResult { string url = 1;
string title = 2; repeated string authors = 3;
} service Searcher { rpc PerformSearch
(SearchRequest) returns (SearchResponse); }

```

The Protobuf style guide recommends using all lowercase with underscores for field names, all uppercase with underscores for enum values, and so on. The C# tooling will automatically convert to .NET styles in the auto-generated types it creates for you. You can read more recommendations at the following link: <https://protobuf.dev/programming-guides/style/>.

Fields must be given a unique number between 1 and 536,870,911. You cannot use the range 19,000 to 19,999 because they are reserved for the Protocol Buffers implementation. These numbers are used instead of the field name during serialization to save space in the binary format.

Good practice: Field numbers cannot be changed once you start using a message because they are tightly bound to the very efficient wire format used by gRPC. Changing a field number is the equivalent of deleting and creating a new field. You should also never reuse a field number. Over time, unused fields remain in gRPC messages because they can no longer be deleted. You can read about the consequences of misusing field numbers at the following link: <https://protobuf.dev/programming-guides/proto3/#consequences>.

Field data types cannot be null, so all number types default to zero (0). Number and other field data types are shown in *Table 14.1*:

Description		
Text values. Defaults to an empty string. Remember that in .NET, string values default to null.		
Boolean values. Defaults to false.		
Variable length encoded 32- and 64-bit integer values. Although they can be used for negative values, it is more efficient to use sint32 or sint64. The C# equivalents to int and long.		
Variable length encoded 32- and 64-bit signed (s) and unsigned (u) integer values. The C# equivalent to int and long, and uint and ulong.		
Always four bytes for 32, or eight bytes for 64. The C# equivalent to uint and ulong, and int and long.		
Floating point real numbers.		
Maximum 2 ³² bytes (4,294,967,296). Use ByteString.CopyFrom(byte[] data) to create a new instance. Use ToByteArray() to get the byte array. Defaults to an empty ByteString value.		

Table 14.1: Number and other field data types in Protobuf

The official guide is found at the following link:
<https://protobuf.dev/programming-guides/proto3/>.

Types of gRPC methods

gRPC has four types of methods, as follows:

- **Unary** methods have structured request and response messages. A unary method completes when the response message is returned. Unary methods should be chosen in all scenarios that do not require a stream. The **Unary** method is the most common because it is simple and structured. The other three methods are all streaming methods that are used when a large amount of data must be exchanged, and they do so by using a stream of bytes. They have the `stream` keyword prefix for either an input parameter, an output parameter, or both.
- **Server streaming** methods receive a request message from the client and return a stream. Multiple messages can be returned over the stream. A server streaming call ends when the server side method returns, but the server side method could run until it receives a cancellation token from the client.
- **Client streaming** methods only receive a stream from the client without any message. The server side method processes the stream until it is ready to return a response message. Once the server side method returns its message, the client streaming call is done.
- **Bi-directional streaming** methods only receive a stream from the client without any message and only return data via a second stream. The call is done when the server side method returns. Once a bi-directional streaming method is called, the client and service can send messages to each other at any time.

In this book, we will only look at the details of unary methods. If you would like

the next edition to cover streaming methods, please let me know.

Microsoft's gRPC packages

Microsoft has invested in building a set of packages for .NET to work with gRPC and, since May 2021, it has been Microsoft's recommended implementation of gRPC for .NET.

Microsoft's gRPC for .NET includes:

- `Grpc.AspNetCore` for hosting a gRPC service in ASP.NET Core.
- `Grpc.Net.Client` for adding gRPC client support to any .NET project by building on `HttpClient`.
- `Grpc.Net.ClientFactory` for adding gRPC client support to any .NET code base by building on `HttpClientFactory`.

You can learn more at the following link:

<https://github.com/grpc/grpc-dotnet>.

Now that we've covered the basic theory of gRPC services, let's see a practical example by building a gRPC service and a gRPC client.

Building a gRPC service and client

Let's see an example service and client for sending and receiving simple messages.

Building a Hello World gRPC service

We will start by building the gRPC service using one of the project templates provided as standard:

1. Use your preferred code editor to create a new project, as defined in the following list:

- Project template: **ASP.NET Core gRPC Service / grpc**
- Solution file and folder: `ModernServices`
- Project file and folder: `Northwind.Grpc.Service`
- **Enable container support:** Cleared
- **Do not use top-level statements:** Cleared
- **Enable native AOT publish:** Selected
- **Enlist in Aspire orchestration:** Cleared

Good practice: Make sure to select **Enable native AOT publish**. With .NET 8 and later, gRPC projects can be **ahead-of-time (AOT)** compiled for native platforms. This gives improved performance and a reduced start time, which is important for microservices that are frequently redeployed and spun up and down during scaling.

1. In the `Protos` folder, in `greet.proto`, note that it defines a service named `Greeter` with a method named `SayHello` that exchanges messages named `HelloRequest` and `HelloReply`, as shown in the following code:

```
syntax = "proto3"; option csharp_namespace
= "Northwind.Grpc.Service"; package greet;
// The greeting service definition.
service Greeter { // Sends a greeting rpc
SayHello (HelloRequest) returns
(HelloReply); } // The request message
containing the user's name. message
HelloRequest { string name = 1; } // The
response message containing the greetings.
```

```
message HelloReply { string message = 1; }
```

For working with .proto files in VS Code, you can install the extension **Protobuf VSC** (drblury.protobuf-vsc). For Rider, you can install the Protocol Buffers plugin from JetBrains, as shown at the following link: <https://plugins.jetbrains.com/plugin/14004-protocol-buffers>.

1. In the Northwind.Grpc.Service.csproj project file, note that this project has native AOT publish enabled, and the .proto file is registered for use on the server side. Then delete the version attribute for the package reference for implementing a gRPC service hosted in ASP.NET Core, as shown highlighted in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup> <TargetFramework>net10.0</
TargetFramework> <Nullable>enable</
Nullable> <ImplicitUsings>enable</
ImplicitUsings>
  <InvariantGlobalization>true</
InvariantGlobalization> <PublishAot>true</
PublishAot> </PropertyGroup> <ItemGroup>
  <Protobuf Include="Protos\greet.proto"
GrpcServices="Server" /> </ItemGroup>
  <ItemGroup> <PackageReference
Include="Grpc.AspNetCore" /> </ItemGroup>
</Project>
```

For Rider, manually add
<PublishAot>true</
PublishAot> if it is missing.

1. Set invariant globalization to `false`, as shown in the following markup:

```
<InvariantGlobalization>false</  
InvariantGlobalization>
```

2. In the `Services` folder, in `GreeterService.cs`, note that it inherits from a class named `GreeterBase` and that it asynchronously implements the `Greeter` service contract by having a `SayHello` method that accepts a `HelloRequest` input parameter and returns a `HelloReply`, as shown in the following code:

```
using Grpc.Core; using  
Northwind.Grpc.Service; namespace  
Northwind.Grpc.Service.Services { public  
class GreeterService : Greeter.GreeterBase  
{ private readonly ILogger<GreeterService>  
_logger; public  
GreeterService(ILogger<GreeterService>  
logger) { _logger = logger; } public  
override Task<HelloReply> SayHello(  
HelloRequest request, ServerCallContext  
context) { return Task.FromResult(new  
HelloReply { Message = "Hello " +  
request.Name }); } }
```

3. If you are using Visual Studio, in **Solution Explorer**, click **Show All Files**. If you are using Rider, then hover over the **Solution** pane and click the eyeball icon.

4. In the `obj\Debug\net10.0\Protos` folder, note the two class files named `Greet.cs` and `GreetGrpc.cs` that are automatically generated from the `greet.proto` file, as shown in *Figure 14.2*:

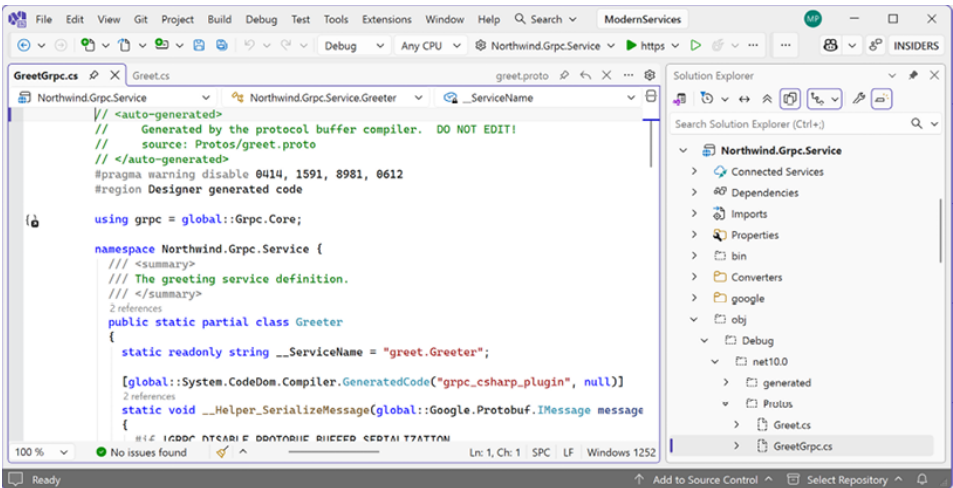


Figure 14.2: The autogenerated class files from a .proto file for a gRPC service

1. In `GreetGrpc.cs`, note the `Greeter.GreeterBase` class that the `GreeterService` class inherited from. You do not need to understand how this base class is implemented, but you should know it is what handles all the details of gRPC's efficient communication.
2. If you are using Visual Studio, in **Solution Explorer**, expand **Dependencies**, expand **Packages**, expand **Grpc.AspNetCore**, and note that it has dependencies on Google's **Google.Protobuf** package, and Microsoft's **Grpc.AspNetCore.Server.ClientFactory** and **Grpc.Tools** packages, as shown in *Figure 14.3*:

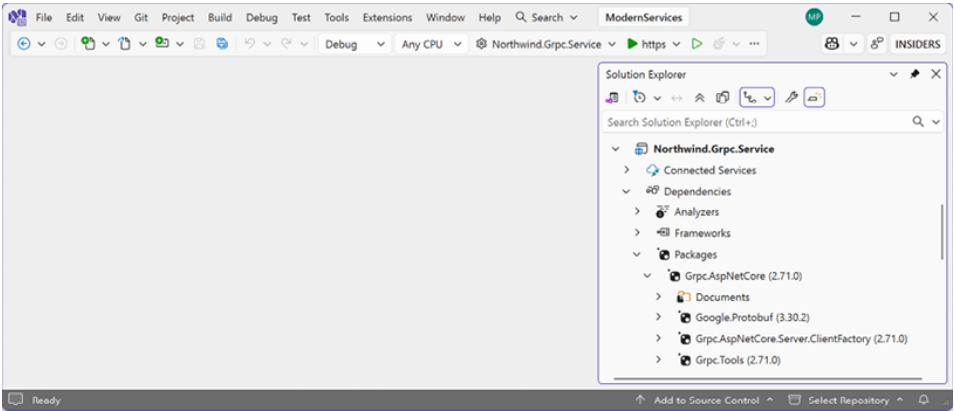


Figure 14.3: The *Grpc.AspNetCore* package references the *Grpc.Tools* and *Google.Protobuf* packages

The `Grpc.Tools` package generates the C# class files from the registered `.proto` files, and those class files use types defined in Google's package to implement the serialization to the Protobuf serialization format. The `Grpc.AspNetCore.Server.ClientFactory` package includes both server side and client-side support for gRPC in a .NET project.

1. In `Program.cs`, in the section that configures services, note the call to add gRPC to the `Services` collection, as shown in the following code:

```
builder.Services.AddGrpc();
```

2. In `Program.cs`, in the section for configuring the HTTP pipeline, note the call to map the `Greeter` service, as shown in the following code:

```
app.MapGrpcService<GreeterService>();
```

3. In the `Properties` folder, open `launchSettings.json` and modify the `applicationUrl` setting to use port 5141 for https and port 5142 for http, as shown highlighted in the following markup:

```
{ "$schema": "http://json.schemastore.org/
launchsettings.json", "profiles": {
"http": { "commandName": "Project",
"dotnetRunMessages": true,
"launchBrowser": false, "applicationUrl"
"http://localhost:5142":,
"environmentVariables": {
"ASPNETCORE_ENVIRONMENT": "Development" }
}, "https": { "commandName": "Project",
"dotnetRunMessages": true,
"launchBrowser": false, "applicationUrl"
"https://localhost:5141;http://
localhost:5142":, "environmentVariables":
{ "ASPNETCORE_ENVIRONMENT": "Development"
} } } }
```

4. Build the `Northwind.Grpc.Service` project.

Project file item configuration syntax

Before we continue, let's quickly review common project file item configuration syntax.

Item configuration generated by Visual Studio commonly uses attributes for the item properties, as shown in the following markup:

```
<Protobuf Include="Protos\greet.proto"  
GrpcServices="Client" />
```

Properties not explicitly set will have their default values.

Item configuration generated by other tools like Rider commonly uses child elements for the item properties, as shown in the following markup:

```
<Protobuf> <Include>Protos\greet.proto</  
Include> <GrpcServices>Client</GrpcServices>  
<Access>Public</Access> <ProtoCompile>True</  
ProtoCompile> <CompileOutputs>True</  
CompileOutputs> <OutputDir>obj\Debug  
\net10.0\</OutputDir>  
<Generator>MSBuild:Compile</Generator>  
<Protobuf>
```

They both achieve the same ends. The first is more concise and recommended for use.

This `<Protobuf>` item tells the gRPC MSBuild tooling how to process a `.proto` file during the build. Under the hood, the `Grpc.Tools` package plugs into MSBuild and runs `protoc` for you, generating C# code and optionally compiling it into your assembly.

`<Include>Protos\greet.proto</Include>` is simply the path to the `.proto` file relative to the project file. MSBuild treats this as an item, not just a filename. Once it's included, the gRPC tooling knows this file should be passed to `protoc`.

`<GrpcServices>Client</GrpcServices>` is one of the most important settings. It controls which gRPC service stubs are generated. Possible values are:

- `Client`, which generates only the client-side code. That means: the

strongly typed gRPC client class, e.g. `Greeter.GreeterClient`, message types, but no server base classes.

- `Server`, which generates server-side base classes, e.g. `Greeter.GreeterBase`.
- `Both`, which generates both client and server code. This is usually lazy configuration. You should split client and server concerns cleanly unless you really need both in the same assembly.
- `None`, which generates only the message types, no gRPC service code at all.

`<Access>Public</Access>` controls the accessibility of the generated C# types. `Public` means generated classes are `public`. `Internal` means generated classes are `internal`.

`<ProtoCompile>True</ProtoCompile>` tells MSBuild whether this `.proto` file should be compiled by `protoc`. This is almost always `True` and is the default if you omit it. You mainly set this to `False` if you want the file included for reference, or you are manually invoking `protoc` yourself, or you only want the file copied somewhere. In normal gRPC .NET projects, explicitly setting this is redundant.

`<CompileOutputs>True</CompileOutputs>` controls whether the generated `.cs` files are compiled into the project. Setting this to `False` is rare, but it can be useful if you want to inspect or post-process generated code, you are generating code for another project, or you are debugging the build pipeline. For everyday use, `True` is correct and usually implicit.

`<OutputDir>obj\Debug\net10.0\</OutputDir>` controls where the generated files go. By default, gRPC tooling puts generated code somewhere under `obj\<Configuration>\<TargetFramework>`. Unless you have a very specific reason, do not override `OutputDir`. Let the tooling handle it.

`<Generator>MSBuild:Compile</Generator>` tells MSBuild how the generated files should be treated. `MSBuild:Compile` means the generated files are treated as C# source files, they are compiled as part of the normal build,

you don't see them in Solution Explorer unless you go digging in `obj`. This is the standard and recommended generator for gRPC in modern .NET projects.

Building a Hello World gRPC client

We will add an ASP.NET Core MVC website project and then add the gRPC client packages to enable it to call the gRPC service:

1. Use your preferred code editor to add a new project, as defined in the following list:
 - Project template: **ASP.NET Core Web App (Model-View-Controller)** / `mvc`
 - Solution file and folder: `ModernServices`
 - Project file and folder:
`Northwind.Grpc.Client.Mvc`
 - **Authentication type**: None
 - **Configure for HTTPS**: Selected
 - **Enable container support**: Cleared
 - **Do not use top-level statements**: Cleared
 - **Enlist in Aspire orchestration**: Cleared
10. In the `Northwind.Grpc.Client.Mvc` project, treat warnings as errors, add package references for Microsoft's gRPC client factory and tools, and Google's .NET library for Protocol Buffers, as shown in the following markup:

```
<ItemGroup> <PackageReference
  Include="Google.Protobuf" />
<PackageReference
  Include="Grpc.Net.ClientFactory" />
<PackageReference Include="Grpc.Tools">
  <PrivateAssets>all</PrivateAssets>
  <IncludeAssets>runtime; build; native;
  contentfiles; analyzers; buildtransitive</
```

```
IncludeAssets> </PackageReference> </ItemGroup>
```

Good practice: The

`Grpc.Net.ClientFactory` package references the `Grpc.Net.Client` package that implements client-side support for gRPC in a .NET project, but it does not reference other packages like `Grpc.Tools` or `Google.Protobuf`. We must reference those packages explicitly. The `Grpc.Tools` package is only used during development, so it is marked as `PrivateAssets=all` to ensure that the tools are not published with the production website.

1. In the `Properties` folder, open `launchSettings.json`, and for the `https` profile, modify the `applicationUrl` setting to use port 5143 for `https` and port 5144 for `http`, as shown highlighted in the following partial markup:

```
"profiles": { ... "https" :{  
  "commandName": "Project",  
  "dotnetRunMessages": true,  
  "launchBrowser": true, "applicationUrl"  
  "https://localhost:5143;http://  
  localhost:5144":, "environmentVariables":  
  { "ASPNETCORE_ENVIRONMENT": "Development"  
  }  
}
```

2. Copy the `Protos` folder from the `Northwind.Grpc.Service`

project/folder to the `Northwind.Grpc.Client.Mvc` project/folder.

In Visual Studio, you can drag and drop to copy. In VS Code or Rider, drag and drop while holding the *Ctrl* or *Cmd* key.

Good practice: In production projects, you would store your `.proto` files in a shared folder (for example, at the solution rather than project level) and then reference them in service and client projects. This means that you only have one set of `.proto` files to modify if you need to make future changes. For the projects in this book, we will keep it simpler.

1. In the `Northwind.Grpc.Client.Mvc` project, in the `Protos` folder, in `greet.proto`, modify the namespace to match the namespace for the current project so that the automatically generated classes will be in the same namespace, as shown in the following code:

```
option csharp_namespace =  
    "Northwind.Grpc.Client.Mvc";
```

2. In the `Northwind.Grpc.Client.Mvc` project file, add or modify the item group that registers the `.proto` file to indicate that it is being used on the client side, as shown highlighted in the following markup:

```
<ItemGroup> <Protobuf Include="Protos  
    \greet.proto" GrpcServices="Client" /> </  
ItemGroup>
```

Visual Studio will have created the item group for you, but it will set `GrpcServices` to `Server` by default, so you must manually change that to `Client`. For other code editors, you might have to create the whole `<ItemGroup>` manually. Rider has more configuration, but you can ignore it.

1. Build the `Northwind.Grpc.Client.Mvc` project to ensure that the automatically generated classes are created.
2. In the `Northwind.Grpc.Client.Mvc` project, in the `obj\Debug\net10.0\Protos` folder, in `GreetGrpc.cs`, note the `Greeter.GreeterClient` class, as partially shown in the following code:

```
public static partial class Greeter { ...  
public partial class GreeterClient :  
    grpc::ClientBase<GreeterClient> {
```

3. In the `Northwind.Grpc.Client.Mvc` project, in `Program.cs`, import the namespace for `Greeter.GreeterClient`, as shown in the following code:

```
using Northwind.Grpc.Client.Mvc; // To use  
Greeter.GreeterClient.
```

4. In `Program.cs`, in the section for configuring services, write a statement to add the `GreeterClient` as a named gRPC client that will be communicating with a service that is listening on port `5141`, as shown in the following code:


```
builder.Services.AddGrpcClient<Greeter.GreeterClient>(
    options => { options.Address = new
Uri("https://localhost:5141"); });
```

5. In the `Models` folder, add a new class named `HomeIndexViewModel.cs`.
6. In `HomeIndexViewModel.cs`, define a class to store a greeting and an error message, as shown in the following code:

```
namespace
Northwind.Grpc.Client.Mvc.Models; public
class HomeIndexViewModel { public string?
Greeting { get; set; } public string?
ErrorMessage { get; set; } }
```

7. In the `Controllers` folder, in `HomeController.cs`, import the namespace to work with the gRPC client factory, as shown in the following code:

```
using Grpc.Net.ClientFactory; // To use
GrpcClientFactory.
```

8. In the `Controller` class, declare a field to store a logger, a field to store a `Greeter Client` instance, and set them by using the client factory in the constructor, as shown highlighted in the following code:

```
public class HomeController : Controller {
private readonly ILogger<HomeController>
_logger; private readonly
Greeter.GreeterClient _greeterClient;
public
```

```
HomeController (ILogger<HomeController>
logger, GrpcClientFactory factory) {
_logger = logger; _greeterClient = factory
.CreateClient<Greeter.GreeterClient> ("Greeter");
}
```

9. In the `Index` action method, make the method asynchronous, add a string parameter named `name` with a default value of `Henrietta`, and then add statements to use the gRPC client to call the `SayHelloAsync` method, passing a `HelloRequest` object and storing the `HelloReply` response in `ViewData`, while catching any exceptions, as shown highlighted in the following code:

```
public async Task<IActionResult> Index(
string name = "Henrietta") {
    HomeIndexViewModel model = new(); try {
        HelloReply reply = await
        _greeterClient.SayHelloAsync( new
        HelloRequest { Name = name });
        model.Greeting = "Greeting from gRPC
        service: " + reply.Message; } catch
        (Exception ex) { _logger.LogWarning(
        $"Northwind.Grpc.Service is not
        responding."); model.ErrorMessage =
        ex.Message; } return View(model); }
```

10. In `Views/Home`, in `Index.cshtml`, after the **Welcome** heading, remove the existing `<p>` element. After that, add markup to render a form for the visitor to enter their name, and then if they submit and the gRPC service responds, to output the greeting, as shown highlighted in the following markup:

```

@using Northwind.Grpc.Client.Mvc.Models
@model HomeIndexViewModel @{
    ViewData["Title"] = "Home Page"; } <div
class="text-center"> <h1
class="display-4">Welcome</h1> <div
class="alert alert-secondary"> <form>
<input name="name" placeholder="Enter your
name" /> <input type="submit" /> </form>
</div> @if (Model.Greeting is not null) {
<p class="alert alert-
primary">@Model.Greeting</p> } @if
(Model.ErrorMessage is not null) { <p
class="alert alert-
danger">@Model.ErrorMessage</p> } </div>

```

If you clean a gRPC project, then you will lose the automatically generated types and see compile errors. To recreate them, simply make any change to a .proto file or close and reopen the project/solution.

Testing a gRPC service and client

Now we can start the gRPC service and see if the MVC website can call it successfully:

1. Start the `Northwind.Grpc.Service` project without debugging.
2. Start the `Northwind.Grpc.Client.Mvc` project without debugging.
3. If necessary, start a browser and navigate to the home page: `https://localhost:5143/`.
4. Note the greeting on the home page, as shown in *Figure 14.4*:

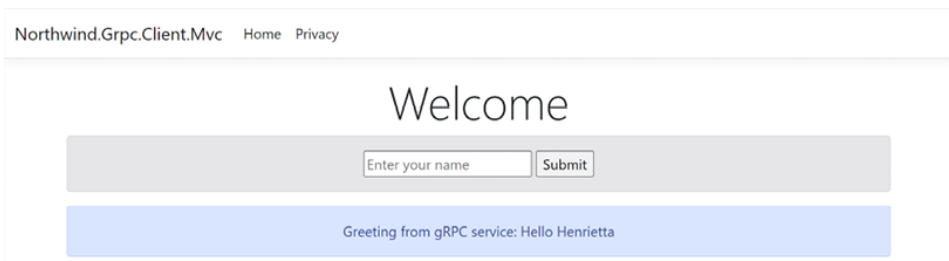


Figure 14.4: Home page after calling the gRPC service to get a greeting

1. View the command prompt or terminal for the ASP.NET Core MVC project and note the info messages that indicate an HTTP/2 POST was processed by the `greet.Greeter/SayHello` endpoint in about 41ms, as shown in the following output:

```
info:
System.Net.Http.HttpClient.Greeter.LogicalHandler
Start processing HTTP request POST
https://localhost:5141/greet.Greeter/
SayHello info:
System.Net.Http.HttpClient.Greeter.ClientHandler[
Sending HTTP request POST https://
localhost:5141/greet.Greeter/SayHello
info:
System.Net.Http.HttpClient.Greeter.ClientHandler[
Received HTTP response headers after
60.5352ms - 200 info:
System.Net.Http.HttpClient.Greeter.LogicalHandler
End processing HTTP request after
69.1623ms - 200
```

2. Enter and submit your own name on the page.
3. Close the browser and shut down the web servers.

Now that you've seen a basic example with simple request and response models,

let's get more realistic by integrating with an EF Core model.

Implementing gRPC for an EF Core model

Now we will add a service for working with the Northwind database to the gRPC project.

Implementing the gRPC service

We will reference the EF Core model for Northwind, then define a contract for the gRPC service using a `.proto` file, and finally implement the service.

We will start with the `Shippers` table because it is simple. Each shipper has only three properties, an `int` and two `string` values, and there are only three records in the table. Let's go:

1. In the `Northwind.Grpc.Service` project, add a project reference to the Northwind database context project, as shown in the following markup:

```
<ItemGroup> <ProjectReference Include="..\
  \..\ModernApps\ Northwind.DataContext
  \Northwind.DataContext.csproj" /> </
ItemGroup>
```

2. Build the `Northwind.Grpc.Service` project.
3. In the `Northwind.Grpc.Service` project, in the `Protos` folder, add a new file (the item template is named **Protocol Buffer File** in Visual Studio) named `shipper.proto`, as shown in the following code:

```
syntax = "proto3"; option csharp_namespace
= "Northwind.Grpc.Service"; package
```

```

shipper; service Shipper { rpc GetShipper
    (ShipperRequest) returns (ShipperReply); }
message ShipperRequest { int32 shipper_id
    = 1; } message ShipperReply { int32
    shipper_id = 1; string company_name = 2;
    string phone = 3; }

```

4. Open the project file and add an entry to include the `shipper.proto` file, as shown highlighted in the following markup:

```

<ItemGroup> <Protobuf Include="Protos
\greet.proto" GrpcServices="Server" />
<Protobuf Include="Protos\shipper.proto"
GrpcServices="Server" /> </ItemGroup>

```

5. Build the `Northwind.Grpc.Service` project.
6. In the `Services` folder, add a new class file named `ShipperService.cs`, and modify its contents to define a shipper service that uses the Northwind database context to return shippers, as shown in the following code:

```

using Grpc.Core; // To use
ServerCallContext. using
Northwind.EntityModels; // To use
NorthwindContext. using ShipperEntity =
Northwind.EntityModels.Shipper; namespace
Northwind.Grpc.Service.Services; public
class ShipperService : Shipper.ShipperBase
{ private readonly ILogger<ShipperService>
_logger; private readonly NorthwindContext
_db; public
ShipperService(ILogger<ShipperService>
logger, NorthwindContext context) {

```

```

_logger = logger; _db = context; } public
override async Task<ShipperReply?>
GetShipper( ShipperRequest request,
ServerCallContext context) {
ShipperEntity? shipper = await
_db.Shippers
.FindAsync(request.ShipperId); return
shipper is null ? null :
ToShipperReply(shipper); } // A mapping
method to convert from a Shipper in the //
entity model to a gRPC ShipperReply.
private ShipperReply
ToShipperReply(ShipperEntity shipper) {
return new ShipperReply { ShipperId =
shipper.ShipperId, CompanyName =
shipper.CompanyName, Phone = shipper.Phone
}; } }

```

The `.proto` file generates classes that represent the messages sent to and from a gRPC service. We therefore cannot use the entity classes defined for the EF Core model. We need a helper method like `ToShipperReply` that can map an instance of an entity class to an instance of the `.proto`-generated classes like `ShipperReply`. This could be a good use for AutoMapper, although in this case, the mapping is simple enough to hand-code.

1. In `Program.cs`, import the namespace for the Northwind database context, as shown in the following code:

```
using Northwind.EntityModels; // To use
AddNorthwindContext method.
```

2. In the section that configures services, add a call to register the Northwind database context, as shown in the following code:

```
builder.Services.AddNorthwindContext();
```

3. In the section that configures the HTTP pipeline, after the call to register GreeterService, add a statement to register ShipperService, as shown in the following code:

```
app.MapGrpcService<ShipperService>();
```

Implementing the gRPC client

Now we can add client capabilities to the Northwind MVC website:

1. Copy the `shipper.proto` file from the `Protos` folder in the `Northwind.Grpc.Service` project to the `Protos` folder in the `Northwind.Grpc.Client.Mvc` project.
2. In the `Northwind.Grpc.Client.Mvc` project, in `shipper.proto`, modify the namespace to match the namespace for the current project so that the automatically generated classes will be in the same namespace, as shown highlighted in the following code:

```
option csharp_namespace =
    "Northwind.Grpc.Client.Mvc";
```

3. In the `Northwind.Grpc.Client.Mvc` project file, modify or add

the entry to register the .proto file as being used on the client side, as shown highlighted in the following markup:

```
<ItemGroup> <Protobuf Include="Protos  
\greet.proto" GrpcServices="Client" />  
<Protobuf Include="Protos\shipper.proto"  
GrpcServices="Client" /> </ItemGroup>
```

If you are using a code editor like Rider that adds extra configuration, I recommend that you simplify the elements as shown in the preceding markup. If you do not, then you might get errors later in this coding task.

1. In the `Northwind.Grpc.Client.Mvc` project file, in `Program.cs`, add a statement to register the `ShipperClient` class to connect to the gRPC service listening on port 5141, as shown in the following code:

```
builder.Services.AddGrpcClient<Shipper.ShipperClient>  
(options => { options.Address = new  
Uri("https://localhost:5141"); });
```

2. In the `Models` folder, in `HomeIndexViewModel.cs`, add a property to store a summary of a shipper, as shown in the following code:

```
public string? ShipperSummary { get; set;  
}
```

3. In the `Controllers` folder, in `HomeController.cs`, declare a field to store a shipper client instance and set it by using the client factory

in the constructor, as shown highlighted in the following code:

```
public class HomeController : Controller {
    private readonly ILogger<HomeController>
    _logger; private readonly
    Greeter.GreeterClient _greeterClient;
    private readonly Shipper.ShipperClient
    _shipperClient; public
    HomeController(ILogger<HomeController>
    logger, GrpcClientFactory factory) {
    _logger = logger; _greeterClient =
    factory.CreateClient<Greeter.GreeterClient>("Greeter");
    _shipperClient =
    factory.CreateClient<Shipper.ShipperClient>("Shipper");
}
```

4. In `HomeController.cs`, in the `Index` action method, add a parameter named `id` and statements to call the `Shipper` gRPC service to get a shipper with the matching `ShipperId`, as shown highlighted in the following code:

```
public async Task<IActionResult> Index(
    string name = "Henrietta", int id = 1) {
    HomeIndexViewModel model = new(); try {
    HelloReply reply = await
    greeterClient.SayHelloAsync( new
    HelloRequest { Name = name });
    model.Greeting = "Greeting from gRPC
    service: " + reply.Message; ShipperReply
    shipperReply = await
    _shipperClient.GetShipperAsync( new
    ShipperRequest { ShipperId = id });
    model.ShipperSummary = "Shipper from gRPC
```

```

service: " + $"ID: {shipperReply.ShipperId
}, Name: {shipperReply.CompanyName },
Phone: {shipperReply.Phone}."; } catch
(Exception ex) { _logger.LogWarning(
$"Northwind.Grpc.Service is not
responding."); model.ErrorMessage =
ex.Message; } return View(); }

```

5. In Views/Home, in Index.cshtml, add code to render a form for the visitor to enter a shipper ID, and render the shipper details after the greeting, as shown highlighted in the following markup:

```

@using Northwind.Grpc.Client.Mvc.Models
@model HomeIndexViewModel @{
    ViewData["Title"] = "Home Page"; } <div
class="text-center"> <h1
class="display-4">Welcome</h1> <div
class="alert alert-secondary"> <form>
<input name="name" placeholder="Enter your
name" /> <input type="submit" /> </form>
<form> <input name="id" placeholder="Enter
a shipper id" /> <input type="submit" />
</form> </div> @if (Model.Greeting is not
null) { <p class="alert alert-
primary">@Model.Greeting</p> } @if
(Model.ErrorMessage is not null) { <p
class="alert alert-
danger">@Model.ErrorMessage</p> } @if
(Model.ShipperSummary is not null) { <p
class="alert alert-
primary">@Model.ShipperSummary</p> } </
div>

```

6. If your database server is not running, for example, because you are

hosting it in Docker, a virtual machine, or in the cloud, then make sure to start it.

7. Start the `Northwind.Grpc.Service` project without debugging.
8. Start the `Northwind.Grpc.Client.Mvc` project.
9. If necessary, start a browser and navigate to the MVC website home page: <https://localhost:5143/>.
10. On the home page, enter a shipper ID (1, 2, or 3) and then click the **Submit** button.
11. Note that an exception is thrown in the gRPC service because the `GetShipper` method uses EF Core, which is attempting to dynamically compile a LINQ query, and that is not supported with native AOT compilation, as shown in the following partial output:

```
fail:
Grpc.AspNetCore.Server.ServerCallHandler[6]
Error when executing service method
'GetShipper'.
System.InvalidOperationException: Model
building is not supported when publishing
with NativeAOT. Use a compiled model.
```

12. Close the browser and shut down the web servers.
13. In the project file, comment out the publish AOT option, as shown in the following markup:

```
<!--<PublishAot>true</PublishAot>-->
```

You might be wondering what the point was of enabling AOT when we created the project and chose to implement parts of

the service using EF Core, if we were just going to have to disable AOT later. There are two reasons: I want you to see the error so you recognize it if you try to do something similar with your own gRPC projects, and hopefully, we *will* be able to use EF Core in the future with .NET 11 or .NET 12.

1. Start the `Northwind.Grpc.Service` project without debugging.
2. Start the `Northwind.Grpc.Client.Mvc` project without debugging.
3. Note the shipper information on the services page, as shown in *Figure 14.5*:

The screenshot shows a web application interface. At the top, the word "Welcome" is displayed in a large, light blue font. Below it, there is a light gray rectangular area containing two input fields and two "Submit" buttons. The first input field is labeled "Enter your name" and the second is labeled "Enter a shipper id". Below these, a light blue rectangular area displays the text "Greeting from gRPC service: Hello Henrietta". At the bottom, another light blue rectangular area displays the text "Shipper from gRPC service: ID: 1, Name: Speedy Express, Phone: (503) 555-9831."

Figure 14.5: Home page after calling the gRPC service to get a shipper

1. There are three shippers in the Northwind database with IDs of 1, 2, and 3. Try entering their IDs to ensure they can all be retrieved, and try entering an ID that does not exist, like 4.
2. Close the browser and shut down the web servers.

So now you've seen gRPC work with simple models and EF Core models. Other features of gRPC that you might want to implement include using native AOT publish, passing metadata, and setting deadlines for improving reliability. Let's have a look at those topics next.

Taking gRPC further

Now let's look at some more advanced topics like native AOT compilation support, getting metadata, adding deadlines, handling dates, times, and decimal types, adding interceptors, and handling exceptions and transient faults.

Improving a gRPC service with native AOT publish

.NET 8 introduced gRPC support for native AOT. But as you have just seen, it is not (yet) compatible with some parts of .NET like EF Core.

Let's change our gRPC service to use the SQL client instead of EF Core. We will leave most of the EF Core code in the project so you can switch back if you want in the future, for example, if you upgrade to EF Core 11 or EF Core 12 if they support native AOT:

1. In the `Northwind.Grpc.Service` project, uncomment out the option to publish AOT and add a package reference for the SQL client, as shown in the following markup:

```
<PackageReference  
  Include="Microsoft.Data.SqlClient" />
```

2. In the `Services` folder, in `ShipperService.cs`, import namespaces for working with `SqlConnection`, as shown in the following code:

```
using Microsoft.Data.SqlClient; // To use  
SqlConnection and so on. using  
System.Data; // To use CommandType.
```

3. In the `GetShipper` method, comment out the statements to get the

shipper from the Northwind data context, and replace them with code to get the shipper using `SqlClient`, as shown highlighted in the following code:

```
public override async Task<ShipperReply?>
GetShipper( ShipperRequest request,
ServerCallContext context) { // We cannot
use EF Core in a native AOT compiled
project. // ShipperEntity? shipper = await
_db.Shippers //
.FindAsync(request.ShipperId);
SqlConnectionStringBuilder builder =
new(); builder.DataSource =
"tcp:127.0.0.1,1433"; // SQL Server in
container. builder.InitialCatalog =
"Northwind";
builder.MultipleActiveResultSets = true;
builder.Encrypt = true;
builder.TrustServerCertificate = true;
builder.ConnectTimeout = 10; // Default is
30 seconds. // builder.DataSource = ".";
// To use local SQL Server. //
builder.IntegratedSecurity = true; // To
use SQL Server Authentication:
builder.UserID = Environment
.GetEnvironmentVariable("MY_SQL_USR");
builder.Password = Environment
.GetEnvironmentVariable("MY_SQL_PWD");
builder.PersistSecurityInfo = false; using
SqlConnection connection =
new(builder.ConnectionString); await
connection.OpenAsync(); using SqlCommand
cmd = connection.CreateCommand();
cmd.CommandType = CommandType.Text;
cmd.CommandText = "SELECT ShipperId,
```

```

CompanyName, Phone" + " FROM Shippers
WHERE ShipperId = @id";
cmd.Parameters.AddWithValue("id",
request.ShipperId); using SqlDataReader r
= await cmd.ExecuteReaderAsync(
CommandBehavior.SingleRow); ShipperReply?
shipper = null; // Read the expected
single row. if (await r.ReadAsync()) {
shipper = new() { ShipperId =
r.GetInt32("ShipperId"), CompanyName =
r.GetString("CompanyName"), Phone =
r.GetString("Phone") }; } await
r.CloseAsync(); return shipper; }

```

4. Double-check that you have re-enabled the publish AOT option.
5. In `Program.cs`, we could alter a statement to use the slim builder for the web application, as shown in the following code:

```

// Use the slim builder to reduce the size
of the application // when using the
publish AOT project option. // var builder
= WebApplication.CreateSlimBuilder(args);

```

The `CreateSlimBuilder` method does not include support for HTTPS or HTTP/3, although you can add those back in yourself if you need them. If we switch to the slim builder, then we must also switch from using HTTPS to HTTP to communicate with the gRPC service. In this task, we will continue to use the “full fat” builder so we can continue to use HTTPS.

1. In the `Northwind.Grpc.Service` project file, add an element to emit compiler-generated files, as shown highlighted in the following markup:

```
<PropertyGroup> <TargetFramework>net10.0</TargetFramework> ...  
<EmitCompilerGeneratedFiles>true</EmitCompilerGeneratedFiles> </PropertyGroup>
```

When

`<EmitCompilerGeneratedFiles>` is `true`, the C# compiler emits files it normally creates behind the scenes. These generated files go into `obj\Debug\netX\GeneratedFiles`. You can open and inspect them to see what the compiler actually produced from your syntax.

1. Build the `Northwind.Grpc.Service` project.
2. If you are using Visual Studio, toggle **Show All Files in Solution Explorer**. If you are using Rider, hover over and then click the eyeball icon.
3. Expand the `obj\Debug\net10.0\generated` folder, and then note the folders and files that have been created by the source generators for AOT and JSON serialization, as shown in *Figure 14.6*:

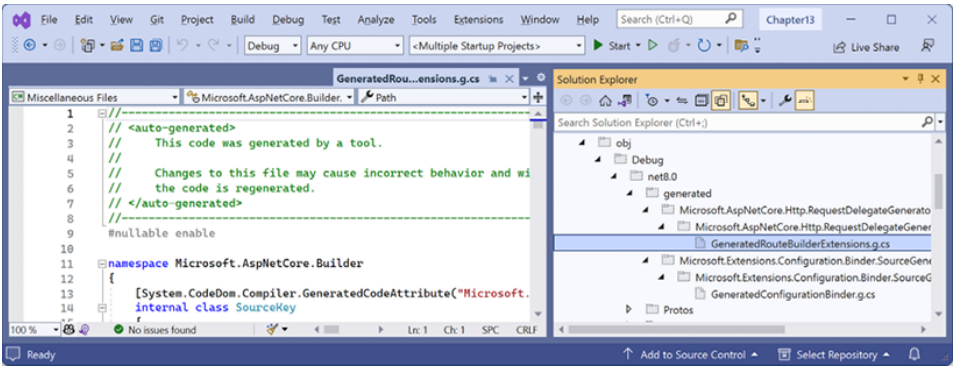


Figure 14.6: Folders and files created by source generators in an AOT gRPC project

1. At the command prompt or terminal, publish the gRPC service using native AOT, as shown in the following command:

```
dotnet publish
```

2. Note the message about generating native code and trim warnings for packages like `Microsoft.Data.SqlClient`, as shown in the following partial output:

```
C:\Users\markj\.nuget\packages
\microsoft.data.sqlclient\6.1.3\runtimes
\win\lib
\net9.0\Microsoft.Data.SqlClient.dll :
warning IL2104: Assembly
'Microsoft.Data.SqlClient' produced trim
warnings. For more information see
https://aka.ms/il2104
```

3. Start **File Explorer** and open the `bin\Release\net10.0\win-x64\publish` folder and note the EXE file is about 47 MB. This and the `Microsoft.Data.SqlClient.SNI.dll` file are the only files

that need to be deployed onto another Windows computer for the web service to work. The `appsettings.json` files are only needed to override configuration if needed. The PDB files are only needed if debugging and, anyway, two of them are only there because we left the EF Core code in the project for reference to make it easier to switch back to non-AOT publishing.

4. Open the `bin\Release\net10.0\win-x64\publish` folder at the command prompt or terminal.
5. At the command prompt or terminal, run `Northwind.Grpc.Service.exe` and explicitly specify the URL with the port number to use, as shown in the following command:

```
Northwind.Grpc.Service.exe --urls  
"https://localhost:5141"
```

The `launchSettings.json` file is only used by code editors like Visual Studio so the ports specified there are ignored and not deployed with the service in production.

1. Start the `Northwind.Grpc.Client.Mvc` project without debugging.
2. Note the web page shows the shipper with an ID of 1 and that you can search for the other shippers.
3. Close the browser and shut down the web servers.

You can learn more about gRPC and native AOT at the following link: <https://learn.microsoft.com/en-us/aspnet/core/grpc/native-aot>.

Getting request and response metadata

Formally defined request and response messages as part of a contract are not the only mechanisms to pass data between a client and service using gRPC. You can also use metadata sent as headers and trailers. Both are simple dictionaries that are passed along with the messages.

Let's see how you can get metadata about a gRPC call:

1. In the `Northwind.Grpc.Client.Mvc` project, in the `Controllers` folder, in `HomeController.cs`, import the namespace to use the `AsyncUnaryCall<T>` class, as shown in the following code:

```
using Grpc.Core; // To use
AsyncUnaryCall<T>.
```

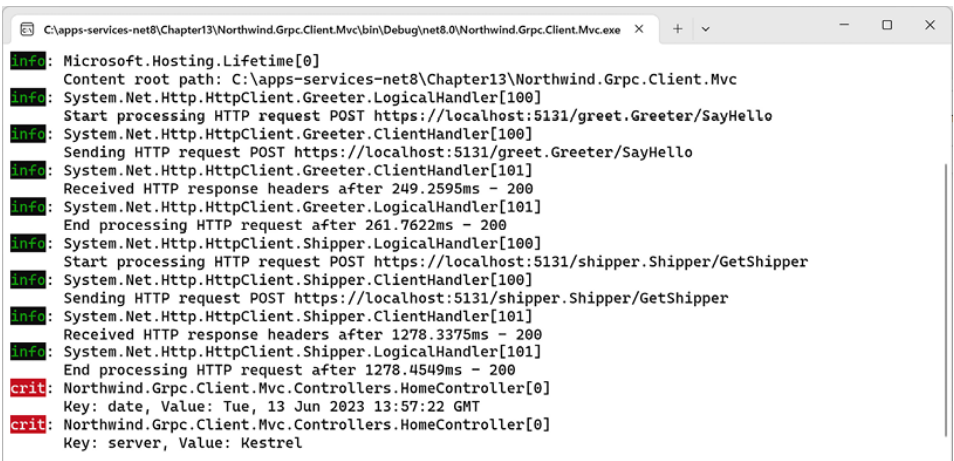
2. In the `Index` method, comment out the statement that makes the call to the gRPC shipper service. Add statements that get the underlying `AsyncUnaryCall<T>` object, then use it to get the headers, output them to the log, and get the response, as shown highlighted in the following code:

```
// ShipperReply shipperReply = await
// _shipperClient.GetShipperAsync( // new
// ShipperRequest { ShipperId = id }); // The
// same call as above but not awaited.
AsyncUnaryCall<ShipperReply> shipperCall =
// _shipperClient.GetShipperAsync( new
// ShipperRequest { ShipperId = id });
Metadata metadata = await
shipperCall.ResponseHeadersAsync; foreach
(Metadata.Entry entry in metadata) { //
// Not really critical, just doing this to
```

make it easier to see.

```
_logger.LogCritical($"Key: {entry.Key},  
Value: {entry.Value}"); } ShipperReply  
shipperReply = await  
shipperCall.ResponseAsync;  
model.ShipperSummary = "Shipper from gRPC  
service: " + $"ID:  
{shipperReply.ShipperId}, Name:  
{shipperReply.CompanyName}, " + $" Phone:  
{shipperReply.Phone}.";
```

3. Start the `Northwind.Grpc.Service` project without debugging.
4. Start the `Northwind.Grpc.Client.Mvc` project without debugging.
5. If necessary, start a browser and navigate to the home page: `https://localhost:5143/`.
6. Note the client successfully making `POST` requests to the `gRPC Greeter` and `Shipper` services and the red critical messages outputting the two entries in the `gRPC metadata` for the call to `GetShipper`, with keys of `date` and `server`, as shown in *Figure 14.7*:



```
info: Microsoft.Hosting.Lifetime[0]  
Content root path: C:\apps-services-net8\Chapter13\Northwind.Grpc.Client.Mvc  
info: System.Net.Http.HttpClient.Greeter.LogicalHandler[100]  
Start processing HTTP request POST https://localhost:5131/greet.Greeter/SayHello  
info: System.Net.Http.HttpClient.Greeter.ClientHandler[100]  
Sending HTTP request POST https://localhost:5131/greet.Greeter/SayHello  
info: System.Net.Http.HttpClient.Greeter.ClientHandler[101]  
Received HTTP response headers after 249.2595ms - 200  
info: System.Net.Http.HttpClient.Greeter.LogicalHandler[101]  
End processing HTTP request after 261.7622ms - 200  
info: System.Net.Http.HttpClient.Shipper.LogicalHandler[100]  
Start processing HTTP request POST https://localhost:5131/shipper.Shipper/GetShipper  
info: System.Net.Http.HttpClient.Shipper.ClientHandler[100]  
Sending HTTP request POST https://localhost:5131/shipper.Shipper/GetShipper  
info: System.Net.Http.HttpClient.Shipper.ClientHandler[101]  
Received HTTP response headers after 1278.3375ms - 200  
info: System.Net.Http.HttpClient.Shipper.LogicalHandler[101]  
End processing HTTP request after 1278.4549ms - 200  
crit: Northwind.Grpc.Client.Mvc.Controllers.HomeController[0]  
Key: date, Value: Tue, 13 Jun 2023 13:57:22 GMT  
crit: Northwind.Grpc.Client.Mvc.Controllers.HomeController[0]  
Key: server, Value: Kestrel
```

Figure 14.7: Logging metadata from a gRPC call

1. Close the browser and shut down the web servers.

The trailers equivalent of the `ResponseHeadersAsync` property is the `GetTrailers` method. It has a return value of `Metadata` that contains the dictionary of trailers. Trailers are accessible at the end of a call.

Adding a deadline for higher reliability

Setting a deadline for a gRPC call is recommended practice because it controls the upper limit of how long a gRPC call can run. It prevents gRPC services from potentially consuming too many server resources.

The deadline information is sent to the service, so the service has an opportunity to give up its work once the deadline has passed, instead of continuing forever. Even if the server completes its work within the deadline, the client may give up before the response arrives at the client because the deadline has passed due to the overhead of communication.

Let's see an example:

1. In the `Northwind.Grpc.Service` project, in the `Services` folder, in `ShipperService.cs`, in the `GetShipper` method, add statements to log the deadline and to pause for five seconds, as shown highlighted in the following code:

```
public override async Task<ShipperReply>
GetShipper( ShipperRequest request,
ServerCallContext context) {
    _logger.LogCritical($"This request has a
deadline of { context.Deadline:T}. It is
now {DateTime.UtcNow:T}."); await
```

```
Task.Delay(TimeSpan.FromSeconds(5)); ... }
```

2. In the `Northwind.Grpc.Service` project, in `appsettings.Development.json`, modify the logging level for ASP.NET Core from the default of `Warning` to `Information`, as shown highlighted in the following configuration:

```
{ "Logging": { "LogLevel": { "Default":  
  "Information", "Microsoft.AspNetCore":  
  "Information" } } }
```

3. In the `Northwind.Grpc.Client.Mvc` project, in the `Controllers` folder, in `HomeController.cs`, in the `Index` method, set a deadline of three seconds when calling the `GetShipperAsync` method, as shown highlighted in the following code:

```
AsyncUnaryCall<ShipperReply> shipperCall =  
    shipperClient.GetShipperAsync( new  
        ShipperRequest { ShipperId = id }, //  
        Deadline must be a UTC DateTime. deadline:  
        DateTime.UtcNow.AddSeconds(3) );
```

4. In `HomeController.cs`, in the `Index` method, before the existing catch block, add a catch block for an `RpcException` when the exception's status code matches the code for deadline exceeded, as shown highlighted in the following code:

```
catch (RpcException rpcex) when  
    (rpcex.StatusCode ==  
    global::Grpc.Core.StatusCode.DeadlineExceeded)
```

```
{
  _logger.LogWarning("Northwind.Grpc.Service
deadline exceeded."); model.ErrorMessage =
rpccex.Message; } catch (Exception ex) {
  _logger.LogWarning($"Northwind.Grpc.Service
is not responding."); model.ErrorMessage =
ex.Message; }
```

5. In the `Northwind.Grpc.Client.Mvc` project, in `appsettings.Development.json`, modify the ASP.NET Core logging level from the default of `Warning` to `Information`, as shown highlighted in the following configuration:

```
{ "Logging": { "LogLevel": { "Default":
"Information", "Microsoft.AspNetCore":
"Information" } } }
```

6. Start the `Northwind.Grpc.Service` project without debugging.
7. Start the `Northwind.Grpc.Client.Mvc` project without debugging.
8. If necessary, start a browser and navigate to the home page: `https://localhost:5143/`.
9. At the command prompt or terminal for the gRPC service, note the request has a three-second deadline, as shown in the following output:

```
crit:
Northwind.Grpc.Service.Services.ShipperService [0]
This request has a deadline of 14:56:30.
It is now 14:56:27.
```

10. In the browser, note that after three seconds, the home page shows a

deadline exceeded exception, as shown in *Figure 14.8*:

Welcome

Enter your name	<input type="submit" value="Submit"/>
Enter a shipper id	<input type="submit" value="Submit"/>

Greeting from gRPC service: Hello Henrietta

Status(StatusCode="DeadlineExceeded", Detail="Deadline Exceeded")

Figure 14.8: A deadline has been exceeded

1. At the command prompt or terminal for the ASP.NET Core MVC client, note the logs that start at the point where a request is made to the `GetShipper` method on the gRPC service, but the deadline is exceeded, as shown in the following output:

```
info:
System.Net.Http.HttpClient.Shipper.LogicalHandler
Start processing HTTP request POST
https://localhost:5141/shipper.Shipper/
GetShipper info:
System.Net.Http.HttpClient.Shipper.ClientHandler
Sending HTTP request POST https://
localhost:5141/shipper.Shipper/GetShipper
warn: Grpc.Net.Client.Internal.GrpcCall[7]
gRPC call deadline exceeded. info:
Grpc.Net.Client.Internal.GrpcCall[3] Call
failed with gRPC error status. Status
code: 'DeadlineExceeded', Message: ''.
```

2. Close the browser and shut down the web servers.
3. In `ShipperService.cs`, comment out the statement that causes a five-second delay, as shown in the following code:

```
// await  
Task.Delay(TimeSpan.FromSeconds(5));
```

Good practice: The default is no deadline. Always set a deadline in the client call. In your service implementation, get the deadline and use it to automatically abandon the work if it is exceeded. Pass the cancellation token to any asynchronous calls so that work completes quickly on the server and frees up resources.

So you've seen some cool features of gRPC like deadlines and native AOT support. You might be surprised to learn that gRPC does not have built-in support for some common data types. Let's see how we can add support for dates, times, and decimal numbers.

Handling dates, times, and decimal numbers

You might have noted that there are no date/time types built into gRPC. To store these values, you must use well-known type extensions, for example, `google.protobuf.Timestamp` (equivalent to `DateTimeOffset`) and `google.protobuf.Duration` (equivalent to `TimeSpan`).

To use them as field types in a message, they must be imported, as shown in the following code:

```
syntax = "proto3"; import "google/protobuf/  
duration.proto"; import "google/protobuf/  
timestamp.proto"; message Employee { int32  
employeeId = 1; google.protobuf.Timestamp
```

```
birth_date = 2; google.protobuf.Duration
earned_vacation_time = 3; ... }
```

The class generated will not use .NET types directly. Instead, there are intermediate types, as shown in the following code:

```
public class Employee { public int EmployeeId;
public Timestamp BirthDate; public Duration
EarnedVacationTime; }
```

There are conversion methods on the types `FromDateTimeOffset`, `ToDateTimeOffset`, `FromTimeSpan`, and `ToTimeSpan`, as shown in the following code:

```
Employee employee = new() { EmployeeId = 1,
BirthDate = Timestamp.FromDateTimeOffset(new
DateTimeOffset( year: 1998, month: 11, day:
30, hour: 0, minute: 0, second: 0, offset:
TimeSpan.FromHours(-5)), EarnedVacationTime =
Duration.FromTimeSpan(TimeSpan.FromDays(15))
}; DateTimeOffset when =
employee.BirthDate.ToDateTimeOffset();
TimeSpan daysoff =
employee.EarnedVacationTime.ToTimeSpan();
```

gRPC also does not natively support decimal values. In the future, that support might be added, but for now, you must create a custom type to represent it. If you choose to do this, then keep in mind that developers on other platforms will have to understand your custom format and implement their own handling for it.

Defining a custom decimal type and using date/time types

Let's add gRPC services for working with products (which have a `UnitPrice` property that is a `decimal`) and employees (which have `HireDate` properties that are `DateTime` values):

1. In the `Northwind.Grpc.Service` project, in the `Protos` folder, add a new file named `decimal.proto`, and add statements to define a message format for safely storing a `decimal` value, as shown in the following code:

```
syntax = "proto3"; option csharp_namespace
= "Northwind.Grpc.Service"; package
decimal; // Example: 12345.6789 -> { units
= 12345, nanos = 678900000 } message
DecimalValue { // To store the whole units
part of the amount. int64 units = 1; // To
store the nano units of the amount
(10^-9). // Must be same sign as units.
sfixed32 nanos = 2; }
```

2. Add a new file named `product.proto`, and add statements to define messages and service methods to get one product, all products, or products that cost a minimum price, as shown in the following code:

```
syntax = "proto3"; option csharp_namespace
= "Northwind.Grpc.Service"; import
"Protos/decimal.proto"; package product;
service Product { rpc GetProduct
(ProductRequest) returns (ProductReply);
rpc GetProducts (ProductsRequest) returns
(ProductsReply); rpc
```

```

GetProductsMinimumPrice
(ProductsMinimumPriceRequest) returns
(ProductsReply); } message ProductRequest
{ int32 product_id = 1; } message
ProductsRequest { } message
ProductsMinimumPriceRequest {
decimal.DecimalValue minimum_price = 1; }
message ProductReply { int32 product_id =
1; string product_name = 2; int32
supplier_id = 3; int32 category_id = 4;
string quantity_per_unit = 5;
decimal.DecimalValue unit_price = 6; int32
units_in_stock = 7; int32 units_on_order =
8; int32 reorder_level = 9; bool
discontinued = 10; } message ProductsReply
{ repeated ProductReply products = 1; }

```

3. Add a new file named `employee.proto`, and modify it to define messages and service methods to get one employee or all employees. Note we must import the Google extension for `timestamp.proto`, as shown in the following code:

```

syntax = "proto3"; option csharp_namespace
= "Northwind.Grpc.Service"; import
"google/protobuf/duration.proto"; import
"google/protobuf/timestamp.proto"; package
employee; service Employee { rpc
GetEmployee (EmployeeRequest) returns
(EmployeeReply); rpc GetEmployees
(EmployeesRequest) returns
(EmployeesReply); } message
EmployeeRequest { int32 employee_id = 1; }
message EmployeesRequest { } message
EmployeeReply { int32 employee_id = 1;

```

```

string last_name = 2; string first_name =
3; string title = 4; string
title_of_courtesy = 5;
google.protobuf.Timestamp birth_date = 6;
google.protobuf.Timestamp hire_date = 7;
string address = 8; string city = 9;
string region = 10; string postal_code =
11; string country = 12; string home_phone
= 13; string extension = 14; bytes photo =
15; string notes = 16; int32 reports_to =
17; string photo_path = 18; } message
EmployeesReply { repeated EmployeeReply
employees = 1; }

```

4. In the project file, add elements to tell the gRPC tool to process the new .proto files, as shown highlighted in the following markup:

```

<ItemGroup> <Protobuf Include="Protos
\greet.proto" GrpcServices="Server" />
<Protobuf Include="Protos\shipper.proto"
GrpcServices="Server" /> <Protobuf
Include="Protos\decimal.proto"
GrpcServices="Server" /> <Protobuf
Include="Protos\product.proto"
GrpcServices="Server" /> <Protobuf
Include="Protos\employee.proto"
GrpcServices="Server" /> </ItemGroup>

```

5. Rebuild the project to make sure the gRPC tool has created the C# classes in the obj\Debug\net10.0\Protos folder, as shown in *Figure 14.9*:

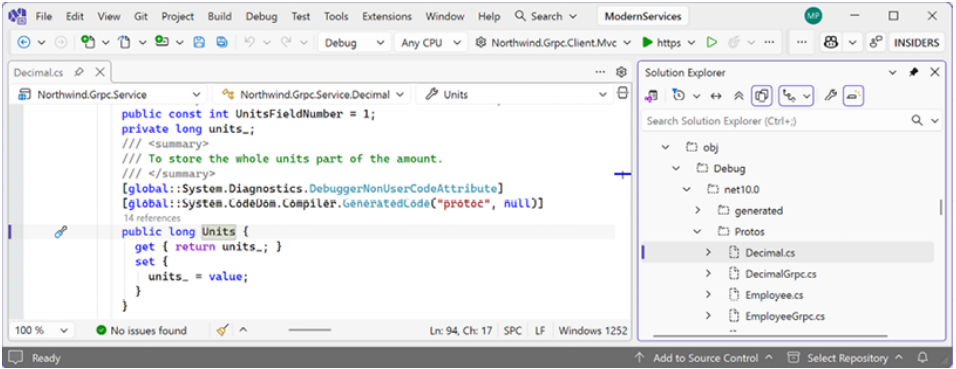


Figure 14.9: gRPC tool-generated classes for a custom decimal type with a Units property

1. In the Northwind.Grpc.Service project, add a new folder named Converters.
2. In the Converters folder, add a new class file named DecimalValue.Converters.cs, and modify its contents to extend the partial class created by the gRPC tools with a constructor and a pair of operators to convert between our custom DecimalValue type and the built-in .NET decimal type, as shown in the following code:

```

namespace Northwind.Grpc.Service; // This
will merge with the DecimalValue type
generated by the // gRPC tools in the obj
\Debug\net10.0\Protos\Decimal.cs file.
public partial class DecimalValue {
    private const decimal NanoFactor =
    1_000_000_000; public DecimalValue(long
    units, int nanos) { Units = units; Nanos =
    nanos; } public static implicit operator
    decimal(DecimalValue grpcDecimal) { return
    grpcDecimal.Units + (grpcDecimal.Nanos /
    NanoFactor); } public static implicit
    operator DecimalValue(decimal value) {
    long units = decimal.ToInt64(value); int
    nanos = decimal.ToInt32((value - units) *

```

```
NanoFactor); return new  
DecimalValue(units, nanos); } }
```

Implementing the product and employee gRPC services

Now we need to implement and register the services:

1. In the `Northwind.Grpc.Service` project, in the `Services` folder, add a new class file named `ProductService.cs` and modify its content to implement the products service. I will leave this as an optional exercise for you, or you can copy the code from the following link: <https://github.com/markjprice/apps-services-net10/blob/main/code/ModernServices/Northwind.Grpc.Service/Services/ProductService.cs>.
2. In the `Services` folder, add a new class file named `EmployeeService.cs` and modify its content to implement the employees service. I will leave this as an optional exercise for you, or you can copy the code from the following link: <https://github.com/markjprice/apps-services-net10/blob/main/code/ModernServices/Northwind.Grpc.Service/Services/EmployeeService.cs>.
3. In `Program.cs`, register the two new services, as shown in the following code:

```
app.MapGrpcService<ProductService>();  
app.MapGrpcService<EmployeeService>();
```

Adding product and employee gRPC

clients

Next, we need to add clients to the MVC project to call the two new gRPC services:

1. In the `Northwind.Grpc.Client.Mvc` project, copy the three `.proto` files from the service project to the MVC project's `Protos` folder.
2. In the three `.proto` files, modify the namespace, as shown in the following code:

```
option csharp_namespace =  
    "Northwind.Grpc.Client.Mvc";
```

3. In the project file, register the three files to create client-side representations, as shown in the following markup:

```
<ItemGroup> <Protobuf Include="Protos  
    \greet.proto" GrpcServices="Client" />  
<Protobuf Include="Protos\shipper.proto"  
    GrpcServices="Client" /> <Protobuf  
    Include="Protos\decimal.proto"  
    GrpcServices="Client" /> <Protobuf  
    Include="Protos\employee.proto"  
    GrpcServices="Client" /> <Protobuf  
    Include="Protos\product.proto"  
    GrpcServices="Client" /> </ItemGroup>
```

If you are using a code editor like Rider that adds extra configuration, I recommend that you simplify the elements as shown in the preceding markup. If you do not, then

you might get errors later in this coding task.

1. Copy the `Converters` folder from the `gRPC` project to the `MVC` project.
2. In the `Converters` folder, in `DecimalValue.Converters.cs`, modify the namespace to use the client, as shown in the following code:

```
namespace Northwind.Grpc.Client.Mvc;
```

3. In the `Northwind.Grpc.Client.Mvc` project, in `Program.cs`, add statements to register clients for the two new services, as shown in the following code:

```
builder.Services.AddGrpcClient<Product.ProductClient>(
    options => { options.Address = new
Uri("https://localhost:5141"); });
builder.Services.AddGrpcClient<Employee.EmployeeClient>(
    options => { options.Address = new
Uri("https://localhost:5141"); });
```

4. In the `Controllers` folder, in `HomeController.cs`, add two fields for the two new clients and set them in the constructor. (Hint: follow the same pattern as for `greeter` and `shipper`.)
5. In `HomeController.cs`, import the namespace for working with repeated fields, as shown in the following code:

```
using Google.Protobuf.Collections; // To
use RepeatedField<T>.
```

6. In `HomeController.cs`, add two action methods for products and

employees, as shown in the following code:

```
public async Task<IActionResult>
Products(decimal minimumPrice = 0M) {
    ProductsReply reply = await
    _productClient.GetProductsMinimumPriceAsync(
    new ProductsMinimumPriceRequest() {
        MinimumPrice = minimumPrice });
    RepeatedField<ProductReply> products =
    reply.Products; return View(products); }
public async Task<IActionResult>
Employees() { EmployeesReply reply = await
    _employeeClient.GetEmployeesAsync( new
    EmployeesRequest());
    RepeatedField<EmployeeReply> employees =
    reply.Employees; return View(employees); }
```

7. In the Views\Shared folder, in _Layout.cshtml, after the menu item for navigating to the home page, add menu items for navigating to products and employees, as shown highlighted in the following markup:

```
<li class="nav-item"> <a class="nav-link
text-dark" asp-area="" asp-
controller="Home" asp-
action="Index">Home</a> </li> <li
class="nav-item"> <a class="nav-link text-
dark" asp-area="" asp-controller="Home"
asp-action="Products">Products</a> </li>
<li class="nav-item"> <a class="nav-link
text-dark" asp-area="" asp-
controller="Home" asp-
action="Employees">Employees</a> </li>
```

8. In the Views\Home folder, add a new Razor View file named Products.cshtml, and modify it to show a table of products, as shown in the following markup:

```
@using Google.Protobuf.Collections @using
Northwind.Grpc.Client.Mvc @model
RepeatedField<ProductReply> @{
    ViewData["Title"] = "Products"; decimal
    price = 0; } <h1>@ViewData["Title"]</h1>
<table class="table table-primary table-
bordered"> <thead> <tr> <th>Product ID</
th> <th>Product Name</th> <th>Unit Price</
th> <th>Units In Stock</th> <th>Units On
Order</th> <th>Reorder Level</th>
<th>Discontinued</th> </tr> </thead>
<tbody> @foreach (ProductReply p in Model)
{ <tr> <td>@p.ProductId</td>
<td>@p.ProductName</td> @{ price =
p.UnitPrice; } <td>@price.ToString("C")</
td> <td>@p.UnitsInStock</td>
<td>@p.UnitsOnOrder</td>
<td>@p.ReorderLevel</td>
<td>@p.Discontinued</td> </tr> } </tbody>
</table>
```

9. In the Views\Home folder, add a new Razor View file named Employees.cshtml, and modify it to show a table of employees, as shown in the following markup:

```
@using Google.Protobuf.Collections @using
Northwind.Grpc.Client.Mvc @model
RepeatedField<EmployeeReply> @{
    ViewData["Title"] = "Employees"; }
```

```

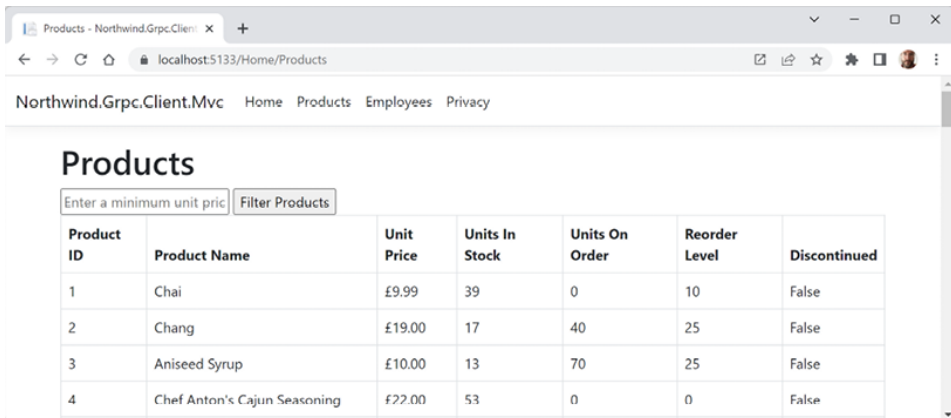
<h1>@ViewData["Title"]</h1> <table
class="table table-primary table-
bordered"> <thead> <tr> <th>Employee ID</
th> <th>Full Name</th> <th>Job Title</th>
<th>Address</th> <th>Birth Date</th>
<th>Photo</th> </tr> </thead> <tbody>
@foreach (EmployeeReply e in Model) { <tr>
<td>@e.EmployeeId</td>
<td>@e.TitleOfCourtesy @e.FirstName
@e.LastName</td> <td>@e.Title</td>
<td>@e.Address<br />@e.City<br /
>@e.Region<br /> @e.PostalCode<br /
>@e.Country</td>
<td>@e.BirthDate.ToDateTimeOffset().ToString("D")
td> <td> </td> </tr> } </tbody> </table>

```

Testing decimal, date, and bytes handling

Finally, we can test the specialized type handling that we implemented:

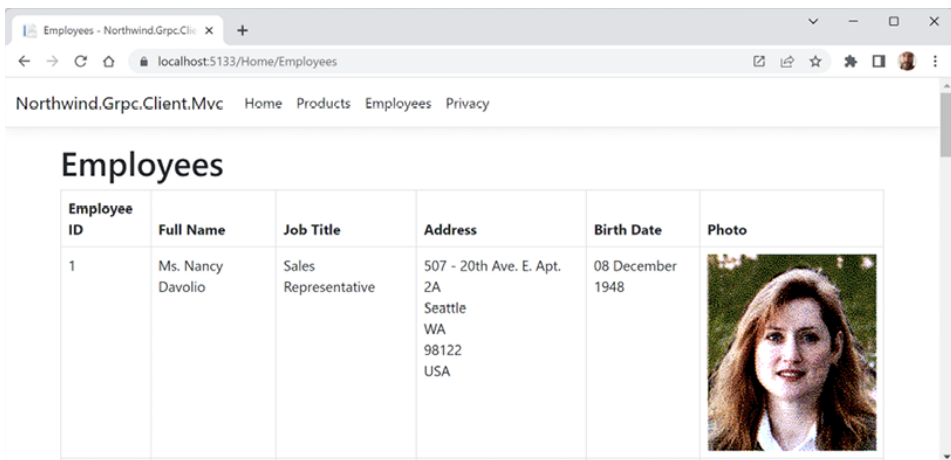
1. Start the `Northwind.Grpc.Service` project without debugging.
2. Start the `Northwind.Grpc.Client.Mvc` project without debugging.
3. On the home page, in the top navigation bar, click **Products**, and note all products are included in the table, as shown in *Figure 14.10*:



Product ID	Product Name	Unit Price	Units In Stock	Units On Order	Reorder Level	Discontinued
1	Chai	£9.99	39	0	10	False
2	Chang	£19.00	17	40	25	False
3	Aniseed Syrup	£10.00	13	70	25	False
4	Chef Anton's Cajun Seasoning	£22.00	53	0	0	False

Figure 14.10: Products including unit prices that use a custom decimal implementation

1. Enter a minimum price such as 1.00, click **Filter Products**, and note that only products with a unit price of that amount or higher are included in the table.
2. In the top navigation bar, click **Employees**, and note that employees and their birth dates and photos are included in the table, as shown in *Figure 14.11*:



Employee ID	Full Name	Job Title	Address	Birth Date	Photo
1	Ms. Nancy Davolio	Sales Representative	507 - 20th Ave. E. Apt. 2A Seattle WA 98122 USA	08 December 1948	

Figure 14.11: Employees including birth dates and photos using timestamp and bytes

1. Close the browser and shut down the web servers.

You've now seen how to use gRPC to build several services that work with data. Now let's see some more advanced features of gRPC.

Implementing interceptors and handling faults

gRPC interceptors are a way to perform additional processing during requests and responses and they can be injected at the client or service. They are often used for logging, monitoring, and validation.

Adding a client-side interceptor

Let's add a client-side gRPC interceptor for logging:

1. In the `Northwind.Grpc.Client.Mvc` project, add a new folder named `Interceptors`.
2. In the `Interceptors` folder, add a new class file named `ClientLoggingInterceptor.cs`, and then add statements to define an interceptor, as shown in the following code:

```
using Grpc.Core.Interceptors; // To use
// Interceptor and so on. using Grpc.Core; //
// To use AsyncUnaryCall<T>. namespace
Northwind.Grpc.Client.Mvc.Interceptors;
public class ClientLoggingInterceptor :
    Interceptor { private readonly ILogger
    _logger; public
    ClientLoggingInterceptor(ILoggerFactory
    loggerFactory) { _logger =
    loggerFactory.CreateLogger<ClientLoggingIntercept
    } public override
    AsyncUnaryCall<TResponse>
    AsyncUnaryCall<TRequest,
    TResponse>(TRequest request,
```

```
ClientInterceptorContext<TRequest,
TResponse> context,
AsyncUnaryCallContinuation<TRequest,
TResponse> continuation) {
_logger.LogWarning("Starting call. Type:
{0}. Method: {1}.", context.Method.Type,
context.Method.Name); return
continuation(request, context); } }
```

Interceptors form a pipeline, so in your interceptor, you must call the next interceptor in the pipeline, represented by the continuation delegate, and pass it the request and context.

1. In the `Northwind.Grpc.Client.Mvc` project, in `Program.cs`, before any of the calls to add gRPC services, add a call to register the interceptor as a singleton service, as shown in the following code:

```
// Register the interceptor before
attaching it to a gRPC client.
builder.Services.AddSingleton<ClientLoggingInterco
```

2. In `Program.cs`, at the end of the statement to register the product client, add the interceptor, as shown highlighted in the following code:

```
builder.Services.AddGrpcClient<Product.ProductCli
options => { options.Address = new
Uri("https://localhost:5141"); })
.AddInterceptor<ClientLoggingInterceptor>();
```


You can attach the logging interceptor to as many clients as you want.

1. Start the `Northwind.Grpc.Service` project without debugging.
2. Start the `Northwind.Grpc.Client.Mvc` project without debugging.
3. On the home page, in the top navigation bar, click **Products**.
4. In the MVC website project command prompt or terminal, note the warning, which will be distinct from information messages as it is yellow-on-black by default, as shown in the following output:

```
warn:
Northwind.Grpc.Client.Mvc.Interceptors.ClientLogging
Starting call. Type: Unary. Method:
GetProductsMinimumPrice.
```

5. Close the browser and shut down the web server.

You might be thinking, “Interceptors sound a lot like ASP.NET Core middleware!” You can read a useful comparison at the following link:

<https://learn.microsoft.com/en-us/aspnet/core/grpc/interceptors#grpc-interceptors-versus-middleware>.

Exception and transient fault handling

gRPC has built-in support to automatically retry failed calls, which is a good way to handle transient faults like temporary network disconnects and down or busy services.

In the client, an `RpcException` could be thrown that includes details of the

error.

First, let's add a transient fault to the gRPC service and see how the client handles it:

1. In the `Northwind.Grpc.Service` project, in the `Services` folder, in `GreeterService.cs`, modify the `SayHello` method to wait for one second and then, randomly, one in three times it should work but two in three times throw a service unavailable exception, as shown in the following code:

```
public override async Task<HelloReply>
SayHello( HelloRequest request,
ServerCallContext context) { await
Task.Delay(1000); // Wait for one second.
if (Random.Shared.Next(1, 4) == 1) // One-
third of the time it works. { return new
HelloReply { Message = "Hello " +
request.Name }; } else // Two-thirds of
the time it fails. { throw new
RpcException(new
Status(StatusCode.Unavailable, "Service is
temporarily unavailable. Try again
later.)); } }
```

2. Start the `Northwind.Grpc.Service` project without debugging.
3. Start the `Northwind.Grpc.Client.Mvc` project without debugging.
4. On the home page, note the exception, as shown in *Figure 14.12*:

Welcome

Enter your name	<input type="submit" value="Submit"/>
Enter a shipper id	<input type="submit" value="Submit"/>

Status(StatusCode="Unavailable", Detail="Service is temporarily unavailable. Try again later.")

Figure 14.12: Service is unavailable exception

1. If you don't get an exception, refresh the page until you do.
2. Close the browser and shut down the web servers.

Adding transient fault handling

Now, let's see how to add transient fault handling to the MVC website client:

1. In the `Northwind.Grpc.Client.Mvc` project, in `Program.cs`, before adding the greeter client to the services collection, add statements to define a `MethodConfig` with a retry policy that retries up to five times for status codes indicating an unavailable service. Then, after configuring the greeter client address, apply the `method config`, as shown highlighted in the following code:

```
MethodConfig configForAllMethods = new() {
    Names = { MethodName.Default },
    RetryPolicy = new RetryPolicy {
        MaxAttempts = 5, InitialBackoff =
        TimeSpan.FromSeconds(1), MaxBackoff =
        TimeSpan.FromSeconds(5), BackoffMultiplier
        = 1.5, RetryableStatusCodes = {
        StatusCode.Unavailable } } };
builder.Services.AddGrpcClient<Greeter.GreeterCli
options => { options.Address = new
Uri("https://localhost:5141"); })
.ConfigureChannel(channel => {
channel.ServiceConfig = new ServiceConfig
```

```
{ MethodConfigs = { configForAllMethods }  
}; });
```

2. Start the `Northwind.Grpc.Service` project without debugging.
3. Start the `Northwind.Grpc.Client.Mvc` project without debugging.
4. On the home page, note the home page might take a few seconds to appear, but eventually, it will successfully appear with the `Hello Henrietta` message from the gRPC service. Also, if you review the gRPC service output, it will include multiple attempts to call `SayHello` before finally working, as shown in the following output:

```
info:  
Microsoft.AspNetCore.Hosting.Diagnostics[1]  
Request starting HTTP/2 POST https://  
localhost:5141/greet.Greeter/SayHello -  
application/grpc - info:  
Microsoft.AspNetCore.Routing.EndpointMiddleware[0]  
Executing endpoint 'gRPC - /greet.Greeter/  
SayHello' info:  
Grpc.AspNetCore.Server.ServerCallHandler[7]  
Error status code 'Unavailable' with  
detail 'Service is temporarily  
unavailable. Try again later.' raised.  
info:  
Microsoft.AspNetCore.Routing.EndpointMiddleware[1]  
Executed endpoint 'gRPC - /greet.Greeter/  
SayHello' info:  
Microsoft.AspNetCore.Hosting.Diagnostics[2]  
Request finished HTTP/2 POST https://  
localhost:5141/greet.Greeter/SayHello -  
200 0 application/grpc 1039.4626ms info:  
Microsoft.AspNetCore.Hosting.Diagnostics[1]
```

```
Request starting HTTP/2 POST https://
localhost:5141/greet.Greeter/SayHello -
application/grpc - info:
Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
Executing endpoint 'gRPC - /greet.Greeter/
SayHello' info:
Grpc.AspNetCore.Server.ServerCallHandler[7]
Error status code 'Unavailable' with
detail 'Service is temporarily
unavailable. Try again later.' raised.
info:
Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
Executed endpoint 'gRPC - /greet.Greeter/
SayHello' info:
Microsoft.AspNetCore.Hosting.Diagnostics[2]
Request finished HTTP/2 POST https://
localhost:5141/greet.Greeter/SayHello -
200 0 application/grpc 1008.1375ms info:
Microsoft.AspNetCore.Hosting.Diagnostics[1]
Request starting HTTP/2 POST https://
localhost:5141/greet.Greeter/SayHello -
application/grpc - info:
Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
Executing endpoint 'gRPC - /greet.Greeter/
SayHello' info:
Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
Executed endpoint 'gRPC - /greet.Greeter/
SayHello' info:
Microsoft.AspNetCore.Hosting.Diagnostics[2]
Request finished HTTP/2 POST https://
localhost:5141/greet.Greeter/SayHello -
200 - application/grpc 1016.4590ms
```

Implementing fault handling is important, but so is supporting web browsers.

Let's look at how gRPC can do that.

Implementing gRPC JSON transcoding

JSON is the most popular format for services that return data to a browser or mobile device. It would be great if we could create a gRPC service and magically make it callable via non-HTTP/2 using JSON.

Thankfully, there is a solution.

Microsoft has a technology they call **gRPC JSON transcoding**. It is an ASP.NET Core extension that creates HTTP endpoints with JSON for gRPC services, based on Google's `HttpRule` class for their gRPC transcoding.

You can read about Google's `HttpRule` class at the following link: <https://cloud.google.com/dotnet/docs/reference/Google.Api.CommonProtos/latest/Google.Api.HttpRule>.

Enabling gRPC JSON transcoding

Let's see how to enable gRPC JSON transcoding in our gRPC service:

1. In the `Northwind.Grpc.Service` project, add a package reference for gRPC JSON transcoding, as shown highlighted in the following markup:

```
<ItemGroup> <PackageReference  
  Include="Grpc.AspNetCore" />  
<PackageReference  
  Include="Microsoft.Data.SqlClient" />  
<PackageReference
```

```
Include="Microsoft.AspNetCore.Grpc.JsonTranscoding"
/> </ItemGroup>
```

2. Build the `Northwind.Grpc.Service` project to restore packages.
3. In `appsettings.json`, modify the `Protocols` option to enable HTTP/1.1 as well as HTTP/2, as shown highlighted in the following markup:

```
{ "Logging": { "LogLevel": { "Default":
"Information", "Microsoft.AspNetCore":
"Warning" } }, "AllowedHosts": "*",
"Kestrel": { "EndpointDefaults": {
"Protocols": "Http1AndHttp2" } } }
```

Good practice: By default, a gRPC project will be configured to only allow HTTP/2 requests. To support clients like `.http` files in your code editor, or Unity, enable both HTTP/1.1 and HTTP/2. Allowing HTTP/1.1 and HTTP/2 on the same port requires TLS for protocol negotiation, which is another good reason to leave HTTPS enabled in a gRPC service and therefore not use `CreateSlimBuilder`.

1. In `Program.cs`, add a call to add JSON transcoding after the call to add gRPC, as shown highlighted in the following code:

```
builder.Services.AddGrpc()
.AddJsonTranscoding();
```

2. In the `Northwind.Grpc.Service` project/folder, add a folder named `google`.
3. In the `google` folder, add a folder named `api`.
4. In the `api` folder, add two `.proto` files named `http.proto` and `annotations.proto`.
5. Copy and paste the raw contents for the two files from the files found at the following link: <https://github.com/dotnet/aspnetcore/tree/main/src/Grpc/JsonTranscoding/test/testassets/Sandbox/google/api>.
6. In the `Protos` folder, in `employee.proto`, import the `annotations.proto` file, and use it to add an option to expose an endpoint to make an HTTP request to the `GetEmployee` method, as shown in the following code:

```
syntax = "proto3"; option csharp_namespace
= "Northwind.Grpc.Service"; import
"google/protobuf/duration.proto"; import
"google/protobuf/timestamp.proto"; import
"google/api/annotations.proto"; package
employee; service Employee { rpc
GetEmployee (EmployeeRequest) returns
(EmployeeReply) { option (google.api.http)
= { get: "/v1/employee/{employee_id}" };
}; rpc GetEmployees (EmployeesRequest)
returns (EmployeesReply); }
```

Testing gRPC JSON transcoding

Now we can start the gRPC service and call it directly from any browser:

1. Start the `Northwind.Grpc.Service` project.
2. Start any browser, show the developer tools, and click the **Network** tab to

start recording network traffic.

3. Navigate to a URL to make a GET request that will call the `GetEmployee` method, `https://localhost:5141/v1/employee/1`, and note the JSON response returned by the gRPC service, as shown in *Figure 14.13*:

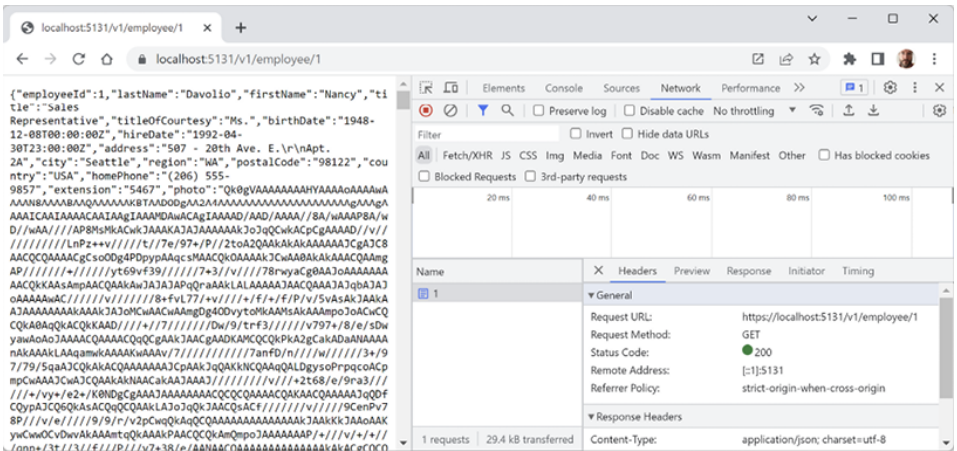


Figure 14.13: Making an HTTP 1.1 GET request to a gRPC service and receiving a response in JSON

1. In your code editor, in the `HttpRequests` folder, create a new file named `grpc-json-transcoding.http`, and add statements to make requests for employees using HTTP/1.1, as shown in the following code:

```
## Configure a variable for the gRPC
service base address. @base_address =
https://localhost:5141/ ## Get Nancy
Davolio. GET {{base_address}}v1/employee/1
## Get Andrew Fuller Davolio. GET
{{base_address}}v1/employee/2
```

2. Send both requests, confirm the responses are correct, and then review the

gRPC service command prompt or terminal to confirm that the requests were made using HTTP/1.1, as shown in the following output:

```
info:
Microsoft.AspNetCore.Hosting.Diagnostics[1]
Request starting HTTP/1.1 GET https://
localhost:5141/v1/employee/2 - - info:
Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
Executing endpoint 'gRPC - /v1/employee/
{employee_id}' info:
Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
Executed endpoint 'gRPC - /v1/employee/
{employee_id}' info:
Microsoft.AspNetCore.Hosting.Diagnostics[2]
Request finished HTTP/1.1 GET https://
localhost:5141/v1/employee/2 - 200 -
application/json;+charset=utf-8 7.9328ms
```

3. Close the .http file, close the browser, and shut down the web server.

Comparing with gRPC-Web

gRPC-Web is an alternative to gRPC JSON transcoding to allow gRPC services to be called from a browser. gRPC-Web achieves this by executing a gRPC-Web client inside the browser. This has the advantage that the communications between the browser and gRPC service use Protobuf and therefore get all the performance and scalability benefits of true gRPC communication.

As you have seen, gRPC JSON transcoding allows browsers to call gRPC services as if they were HTTP APIs with JSON. The browser needs to know nothing about gRPC. The gRPC service is responsible for converting those HTTP API calls into calls to the actual gRPC service implementation.

To simplify and summarize:

- gRPC JSON transcoding happens on the server side.
- gRPC-Web happens on the client side.

Good practice: Add gRPC JSON transcoding support to all your gRPC services hosted in ASP.NET Core. This provides the best of both worlds. Clients that cannot use gRPC natively can call the Web API. Clients that can use gRPC natively can call it directly.

Practicing and exploring

Test your knowledge and understanding by answering some questions, getting some hands-on practice, and exploring this chapter's topics with deeper research.

Exercise 14.1 – Online material

Online material can be extra content written by me for this book, or it can be references to content created by Microsoft or third parties.

Compare gRPC services with HTTP APIs

Review the article found at the following link: <https://learn.microsoft.com/en-us/aspnet/core/grpc/comparison>.

Exercise 14.2 – Test your knowledge

Answer the following questions:

1. What are three benefits of gRPC that make it a good choice for implementing services?
2. How are contracts defined in gRPC?
3. Which of the following .NET types require extensions to be imported:

int, double, or DateTime?

4. Why should you set a deadline when calling a gRPC method?
5. Why should you be cautious if implementing your own custom type to handle decimal values?
6. What do you have to do to use date and time values in a gRPC message?
7. What are some scenarios when you might implement an interceptor?
8. What are the benefits of enabling gRPC JSON transcoding to a gRPC service hosted in ASP.NET Core?

Appendix A, Answers to the Test Your Knowledge Questions, is available to download from the following link: https://github.com/markjprice/apps-services-net10/blob/main/docs/B31467_Appendix%20A.pdf.

Exercise 14.3 – Explore topics

Use the links on the following page to learn more details about the topics covered in this chapter: <https://github.com/markjprice/apps-services-net10/blob/main/docs/book-links.md#chapter-14---building-efficient-microservices-using-grpc>.

Summary

In this chapter, you:

- Learned about some concepts of gRPC services, how they work, and their benefits.
- Implemented a simple gRPC service.
- Implemented a gRPC service that uses an EF Core model that cannot yet use AOT publish.

- Implemented a gRPC service that uses `SqlClient` libraries that can use AOT publish and are therefore smaller and faster.
- Learned how to set deadlines and read metadata sent as headers and trailers.
- Implemented a custom `decimal` type and used extended date/time types.
- Implemented a client-side interceptor.
- Extended a gRPC service with support for being called as an HTTP service with JSON, to support clients that cannot work with gRPC natively.

In the next chapter, you will learn how to continue your learning journey with apps and services for .NET.

Get This Book **UNLOCK NOW** and Exclusive

Scan the QR code
book by name, co



Search for this
ps on the page.

Note: Keep your invoice handy. Purchases made directly from Packt don't require one.

Epilogue

I wanted this book to be different from the others on the market. I hope that you found it to be a brisk, fun read, packed with practical, hands-on walkthroughs of each subject.

This epilogue contains the following short sections:

- Introducing the Survey Project Challenge
- Next steps on your C# and .NET learning journey
- Good luck!

Introducing the Survey Project Challenge

To help us learn all the different technologies that .NET developers need to know these days, it would be great to attempt to implement a project that needs a mix of technologies and skills. It should be as real as possible, be cool, fun, and practical, and have already been implemented by others with public products that we can be inspired by.

The Survey Project Challenge is an optional complete project with a common set of problems that will give you a real-world set of projects to build. You can read more about it at the following link: <https://github.com/markjprice/apps-services-net10/blob/main/docs/ch15-survey-project.md>.

Next steps on your C# and .NET learning journey

There is never enough space in a book written for print to include everything one might want. For subjects that you want to learn more about, I hope that the notes, good practice tips, and links in the GitHub repository point you in the right direction: <https://github.com/markjprice/apps-services-net10/blob/main/docs/book-links.md>.

Companion books to continue your learning journey

This is one of four books in a quartet that provides a learning journey through .NET 10:

1. The first book, *C# 14 and .NET 10 - Modern Cross-Platform Development Fundamentals*, covers the fundamentals of the C# language, the .NET libraries, and modern ASP.NET Core for web development. It is designed to be read linearly because skills and knowledge from earlier chapters build up and are needed to understand later chapters.
2. The second book, *Real-World Web Development with .NET 10*, covers mature and proven web development technologies like ASP.NET Core MVC and controller-based Web API web services, as well as OData, FastEndpoints, and Umbraco CMS for building real-world web projects on .NET 10. You will learn how to test your web services using xUnit and test the user interfaces of your websites using Playwright, and then how to containerize your projects ready for deployment.
3. The third book, *Apps and Services with .NET 10* (the one you're reading now), covers how to build graphical user interfaces for websites, desktop, and mobile apps with Blazor, Avalonia, and .NET MAUI respectively. Then you will learn about more specialized library topics like internationalization, and popular third-party packages including Serilog and NodaTime. You will also learn how to build native AOT-compiled services with ASP.NET Core Minimal API and how to improve performance, scalability, and reliability using caching, queues, and

background services. Finally, you will implement more services using GraphQL, gRPC, and SignalR, as well as learn how to integrate LLMs to add intelligence to your solutions.

4. The fourth book, *Tools and Skills for .NET 10*, covers important tools and skills that a professional .NET developer should have. These include design patterns and solution architecture, debugging, memory analysis, all the important types of testing, whether unit, performance, or web testing, and then containerization for deployment topics like Docker and Aspire. Finally, we look at how to prepare for an interview to get the .NET developer career that you want.

A summary of the .NET 10 quartet and their most important topics is shown in *Figure E.1*:

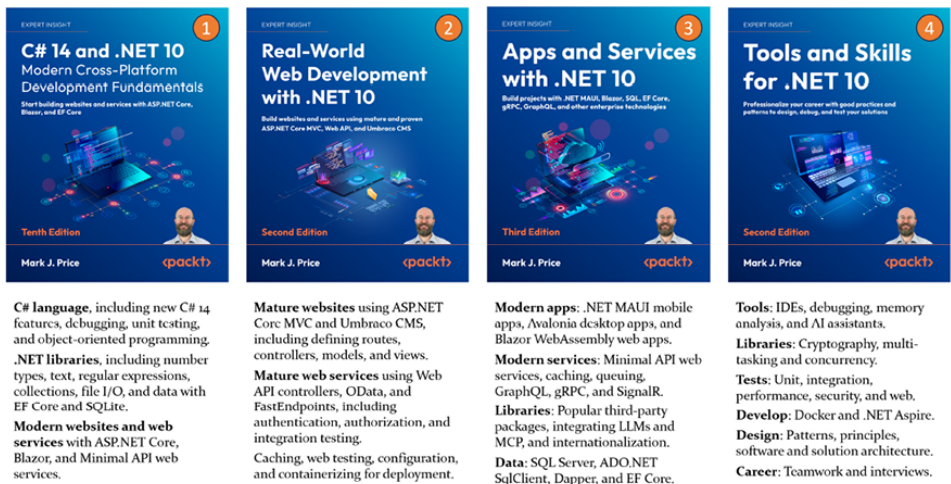


Figure E.1: Companion books for learning .NET 10

To see a list of all the books that I have published with Packt, you can use the following link: <https://subscription.packtpub.com/search?query=mark+j.+price>.

Other books to take your learning further

If you are looking for other books from my publisher that cover related subjects, there are many to choose from, as shown in *Figure E.2*:

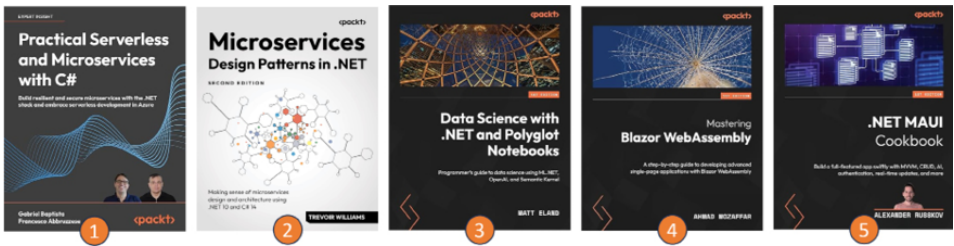


Figure E.2: Packt books to take your apps and services with .NET learning further

Here’s a brief description of each of these five books:

- 1. Practical Serverless and Microservices with C# (July 2025):** Written for .NET developers entering the world of modern cloud and distributed applications, it shows you when microservices and serverless architectures are the right choice for building scalable enterprise solutions and when they’re not. You’ll gain a realistic understanding of their use cases and limitations. Rather than promoting microservices as a one-size-fits-all solution, it encourages thoughtful adoption based on real-world needs. <https://www.amazon.com/dp/B0DNSLZRB1/>
- 2. Microservices Design Patterns in .NET (December 2025):** Making a microservice-based app is no easy feat, and there are many concerns that need to be addressed. As you progress through the chapters of this guide, you’ll dive headfirst into the problems that come packed with this architectural approach, and then explore the design patterns that address these problems. You’ll also learn how to be deliberate and intentional in your architectural design to overcome major considerations in building microservices. <https://www.amazon.com/dp/1837027412/>
- 3. Data Science with .NET and Polyglot Notebooks (August 2024):** Expand your skillset by learning how to perform data science, machine learning, and generative AI experiments in .NET Interactive notebooks using a variety of languages, including C#, F#, SQL, and PowerShell. <https://www.amazon.com/dp/B0D4L1HNMK/>

4. **Mastering Blazor WebAssembly (August 2023):** A complete resource that teaches you everything you need to build client-side web applications. You'll discover the anatomy of a Blazor WebAssembly project, along with the build, style, and structure of the components. You'll implement forms to catch user input and collect data, as well as explore the topics of navigating between the pages in depth. <https://www.amazon.com/dp/B0B2PWQBNM/>
5. **.NET MAUI Cookbook (December 2024):** Build robust cross-platform apps with practical recipes covering UI best practices and performance optimization to authentication, offline data synchronization, and AI integration. <https://www.amazon.com/dp/B0DHY34WQ5/>

Good luck!

I wish you the best of luck with all your .NET projects!

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow this QR code:

https://packt.link/apps_and_services_dotnet10



Join .NETPro – It's Free

Staying sharp in .NET takes more than reading release notes. It requires real-world tips, proven patterns, and scalable solutions. That's what .NETPro, Packt's new newsletter, is all about.

Scan the QR code or visit the link to subscribe:

<https://landing.packtpub.com/dotnetpronewsletter/>



Share your thoughts

Now you've finished *Apps and Services with .NET 10, Third Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here to go straight to the Amazon review page](#) for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Unlock Your Exclusive Benefits

Your copy of this book includes the following exclusive benefits:



DRM-Free PDF with Appendices

Download DRM-free PDF and ePub copies of this book including appendices.



7-Day Packt Library Access

Get 7-day unlimited access to 8,000+ books and videos. No credit card required.

Available for first-time Packt+ trial users only.



Next-Gen Reader Access

Read this book on Packt Reader with progress sync, dark mode and note-taking.

Follow the guide below to unlock them. The process takes only a few minutes and needs to be completed once.

Unlock this Book's Free Benefits in 3 Easy Steps

Step 1

Keep your purchase invoice ready for *Step 3*. If you have a physical copy, scan it using your phone and save it as a PDF, JPG, or PNG.

For more help on finding your invoice, visit <https://www.packtpub.com/en-us/unlock?step=1>.

Note: If you bought this book directly from Packt, no invoice is required. After *Step 2*, you can access your exclusive content right away.

Step 2

Scan the QR code or go to packtpub.com/unlock.



On the page that opens (similar to *Figure U.1* on desktop), search for this book by name and select the correct edition.

Unlock Your Book's Free Benefits

Bought a Packt book from Amazon or one of our channel partners? Unlock your free benefits in 3 easy steps.

Find Your Book

Sign Up or Sign In

Upload Purchase Proof

[Need Help?](#)

✦ 1. Find Your Book

Search by title or ISBN

👤 2. Sign up (Free) or Sign In

📎 3. Upload Purchase Proof

Figure U.1: Packt unlock landing page on desktop

Step 3

After selecting your book, sign in to your Packt account or create one for free. Then upload your invoice (PDF, PNG, or JPG, up to 10 MB). Follow the on-screen instructions to finish the process.

Need Help

If you get stuck and need help, visit <https://www.packtpub.com/unlock-benefits/help> for a detailed FAQ on how to find your invoices and more. This QR code will take you to the help page.



Note: If you are still facing issues, reach out to customercare@packt.com.

Index

Symbols

.NET [5](#)

using, to build MCP server [400-403](#)

.NET 10

platforms for deployment [16](#)

.NET client

testing [631](#), [632](#)

.NET console app SignalR client

.NET client, creating for SignalR [578-580](#)

building [578](#)

testing [581](#), [582](#)

.NET Desktop Runtime [9](#)

.NET MAUI [84](#), [85](#)

handlers [89](#)

mobile and desktop apps, building [90](#)

namespaces [81](#), [82](#)

platform-specific code, writing [89](#), [90](#)

version support [85](#)

workloads, installing manually [86](#)

.NET MAUI controls [83](#), [84](#)

.NET MAUI (Multi-platform App UI) [79](#)

.NET MAUI project

content pages, implementing [102](#), [103](#)

creating [92-95](#)

shell navigation and content pages, adding [96-101](#)

.NET MAUI resource dictionaries

reference link [114](#)

.NET MAUI user interface component categories [87](#)

Editor control [88](#)

Entry control [88](#)

ListView control [88](#)

Shell control [87](#)

.NET Multi-platform App UI (MAUI) [10](#), [11](#)

A

ADO.NET

console app, creating for working with [312-318](#)

isolation levels, in transactions [330](#)

objects, generating with data reader [332-334](#)

streams, outputting with data reader [330](#), [331](#)

transactions, handling with using blocks [329](#)

transactions, key components [328](#), [329](#)

transactions, using [330](#)

types [311](#), [312](#)

used, for execution of stored procedures [323-326](#)

used, for managing transactions [327](#)

used, for query execution [319-321](#)

used, for working with data readers [319-321](#)

working with asynchronously [323](#)

ADO.NET driver for SQL Server [310](#)

Advanced Message Queuing Protocol (AMQP) [526](#)

Agent Framework [385](#)

- AI Chat Web App project template, installing [385](#), [386](#)
- chat app [389](#), [390](#)
- chat app, building [386-388](#)
- chat app, with custom functions [396-398](#)
- functions [390](#), [391](#)
- functions, adding [391-395](#)
- reference link [385](#)
- using, with OpenAI model [384](#)

ahead-of-time (AOT) [664](#)

AI Chat Web App project template [385](#), [386](#)

AI for coding

- references [73](#)

alert component

- building [180](#), [182](#)

Amazon Web Services (AWS) [191](#)

Amazon Web Services (AWS) Lambdas [8](#)

Android [10](#)

annotation attributes [343](#)

AOT warnings

- reference link [475](#)

Apple silicon Macs

- QuestPDF, using [234](#)

app models [5](#)

AppThemeBinding extension [109](#)

app user interface technologies [3](#), [5](#)

artificial intelligence (AI) tools [69](#)

- ChatGPT [69](#), [70](#)

GitHub Copilot [70](#), [71](#)

ASP.NET Core [5](#)

caching guidelines [497](#)

controller-based Web API service, building [497-502](#)

object cache [512](#)

objects, caching with distributed caching [506-510](#)

objects, caching with in-memory caching [503-506](#)

used, for caching [497](#)

web responses, caching with HTTP caching [512-516](#)

ASP.NET Core [9](#) [428](#)

ASP.NET Core Minimal API [429](#)

MapGroup [432](#)

MapWhen [431](#)

OpenAPI document, improving [435](#)

OpenAPI, used for documentation [443-445](#)

parameter mapping [433](#), [434](#)

path exclusion, from OpenAPI documentation [451](#)

return values [434](#)

route mappings [430](#), [431](#)

Scalar, implementing for documentation [445](#), [446](#)

used, for building web services [429](#)

web services, benefits [429](#), [430](#)

web services, testing with code editor tools [446-450](#)

ASP.NET Core Minimal API project

designing [435](#)

setting up [436-442](#)

ASP.NET Core MVC [5](#)

ASP.NET Core MVC project

using, as GraphQL client [623-628](#)

ASP.NET Core Razor Pages [5](#)

ASP.NET Core SignalR [562](#)

assets

adding, to project [145](#)

images, adding to project [146](#), [147](#)

referencing, like images [144](#)

referencing, to project [145](#), [146](#)

Asynchronous JavaScript and XML (AJAX) [562](#)

AutoMapper [213](#)

implementation [213](#)

reference link [220](#), [222](#)

URL [220](#)

AutoMapper configuration

mappers, defining [215](#)

models, defining [213-215](#)

tests, performing [216](#), [218](#)

Avalonia [10](#), [11](#), [119](#), [120](#)

code editor, setting up [124](#)

developer experience [120](#)

project templates, installing [122](#)

reference link [120](#)

Skia graphics engine [121](#)

URL [11](#), [21](#)

Avalonia controls

events, handling [125](#), [126](#)

laying out [126](#)

locating [125](#)

panels, with attached layout properties [130](#)

styling [131](#)

working with [124](#)

Avalonia controls, laying out

example [128](#)

HorizontalAlignment property [126](#)

Margin property [127](#)

Padding property [128](#)

VerticalAlignment property [127](#)

Avalonia Cross Platform Application (avalonia.xplat) [124](#)

Avalonia .NET App (avalonia.app) [123](#)

Avalonia .NET MVVM App (avalonia.mvvm) [123](#)

Avalonia project

creating [134](#), [135](#)

desktop app, implementing for data [139-143](#)

DockPanel [138](#)

DockPanel, for layout in desktop apps [136-138](#)

typical menu, adding [143](#), [144](#)

Avalonia project template installation [122](#), [123](#)

Avalonia Cross Platform Application (avalonia.xplat) [124](#)

Avalonia .NET App (avalonia.app) [123](#)

Avalonia .NET MVVM App (avalonia.mvvm) [123](#)

Avalonia XAML [120](#)

Azure Functions [6](#)

Azure OpenAI model

reference link [380](#)

Azure SignalR Service [563](#)

B

bi-directional streaming method [663](#)

Blazor [5](#), [11](#), [151](#), [152](#)

- base component classes [158](#), [159](#)
- components and routing [153](#), [155](#)
- CSS and JavaScript isolation [161](#)
- hosting models [152](#), [153](#)
- layouts [159](#)
- navigating, to page components [160](#)

Blazor components

- alert component, building [180](#), [182](#)
- Bootstrap icons, using [168](#), [169](#)
- building [161](#)
- building, with server interactivity [172-174](#)
- data context, registering [169](#), [170](#)
- dialog box component, building [175-179](#)
- EF Core class library, referencing [169](#), [170](#)
- progress bar component, building [174](#), [175](#)
- project template, reviewing [161-168](#)
- static server rendered component, building for data [170](#), [172](#)

Blazor data component

- building [182](#), [183](#)
- building, as routable page component [183](#)
- entities, obtaining into [184](#), [185](#)

Blazor Full Stack [153](#)

Blazor Hybrid/.NET MAUI Blazor App [152](#)

Blazor routing

- comparing, with MVC routing [156](#)

- constraints, for parameters [158](#)

- parameters [156](#)

- parameters, setting for query string [157](#)

- to page components [155](#)

Blazor Server [152](#)

Blazor WebAssembly [151](#), [152](#)

Bootstrap icons

- using [168](#), [169](#)

boxing [15](#)

broadcast real-time communication [8](#)

C

C# [18](#)

calculated properties

- on entity creation [354-357](#)

C# Dev Kit [18](#)

C# Dev Kit v1.90.2 [20](#)

Central Package Management (CPM) [23](#)

- benefits [23](#), [24](#)

- configuring [25](#), [33](#)

- features [23](#), [24](#)

- good practice [34](#)

- reference link [35](#)

change tracking [359](#)

chat app [389](#), [390](#)

- trying out, with custom functions [396-398](#)

chat forums [67](#)

ChatGPT [69](#), [70](#)

ChilliCream [595](#)

Circuit Breaker pattern [517](#)

class libraries

creating, for database context [53-58](#)

creating, for entity models [51-53](#)

testing, with xUnit [60-63](#)

class-to-table mapping

improving [59](#)

client streaming method [662](#)

cloud development tools [85](#)

Comet

reference link [10](#)

connection string [301](#)

console app

creating, for working with ADO.NET [312-318](#)

Noda Time [291-294](#)

used, for exploring text manipulations [198-202](#)

used, for message consumption from queue [535-539](#)

console app client

creating, with Strawberry Shake [633-637](#)

content delivery network (CDN) [152](#), [564](#)

Content Management System (CMS) [503](#)

content pages

adding [96-101](#)

implementing [102](#), [103](#)

controller-based Web API service

building [497-502](#)

Coordinated Universal Time (UTC) [243](#)

Copilot+ PCs [418](#)

reference link [418](#)

CoreWCF [7](#)

CreateSlimBuilder method [684](#)

Cross-Origin Resource Sharing (CORS) [461](#), [462](#)

.NET client, creating [457-460](#)

enabling, for specific endpoints [462](#)

HTTP logging, configuring for web services [452](#), [453](#)

policy options [463](#)

used, for relaxing origin security policy [451](#)

web page JavaScript client, creating [453-457](#)

cross-platform desktop apps

building [10](#)

cross-platform mobile apps

building [10](#)

CSS and JavaScript isolation [161](#)

cultures

changing [270-276](#)

detecting [270-276](#)

globalization and localization, testing [282-288](#)

resources, loading [278-281](#)

user interface, localizing [278](#)

working with [269](#), [270](#)

custom decimal type

defining [692-696](#)

D

Dapper

connection extension methods [334](#), [335](#)

data, managing with [334](#)

used, for querying [335-337](#)

data

built-in validators, reviewing [227](#)

model and validator, defining [227-229](#)

validating [227](#)

validation messages, customizing [227](#)

validator, testing [230-233](#)

database context

class library, creating for [53-58](#)

databases [4](#)

database system

cons [12](#), [13](#)

considerations [12](#)

pros [12](#), [13](#)

data binding [114](#), [116](#)

data context

registering [169](#), [170](#)

Data Definition Language (DDL) [309](#)

Data Manipulation Language (DML) [306](#), [307](#)

used, for data addition and update [308](#), [309](#)

used, for data deletion [308](#)

data reader

ADO.NET, used for working with [319-321](#)

objects, generating with [332-334](#)

used, for outputting streams [330](#), [331](#)

data seeding [345](#)

data streaming, with SignalR [582](#)

.NET console app client, creating [584](#), [585](#)

hub, defining for [583](#), [584](#)

testing [587](#), [588](#)

data transfer objects (DTOs) [213](#)

DateOnly

working with [256](#)

dates and times

calculations [250](#)

DayOfWeek enum, localizing [255](#)

DST, complexities [254](#)

formatting information, getting [257](#), [258](#)

globalization [252](#), [253](#)

microseconds [251](#)

nanoseconds [251](#)

values, formatting [247-249](#)

values, specifying [244-247](#)

working with [243](#), [244](#)

date/time types

handling [691](#), [692](#)

using [692-696](#)

Daylight Saving Time (DST) [243](#)

complexities [254](#)

DayOfWeek enum

localizing [255](#), [256](#)

decimal numbers

handling [691](#), [692](#)

DELETE [426-428](#)

denial of service (DoS) [463](#)

rate-limited console client, creating [468-473](#)

rate limiting, with ASP.NET Core middleware [474](#)

rate limiting, with AspNetCoreRateLimit package [464-467](#)

developer experience, Avalonia

Linux and WebAssembly targets [121](#)

Skia-based rendering [121](#)

development environment

setting up [14](#)

dialog box component

building [175-179](#)

direct exchange [526](#)

Discord [67](#)

distributed caching

used, for caching objects [506-510](#)

distributed DoS (DDoS) [463](#)

Distributed Memory Cache [507](#)

Docker

installing [40](#), [41](#)

used, for setting up RabbitMQ [527](#), [528](#)

Docker Command-line Interface (CLI) [40](#)

Docker containers [40](#)

Docker Engine [40](#)

docker ps command

reference link [44](#)

Docker resources

removing [50](#)

Document Object Model (DOM) [152](#)

dotnet-ef command-line tool [50](#)

dotnet tool [65](#), [66](#)

drblury.protobuf-vsc [19](#)

dynamic HTML (DHTML) [561](#)

E

Editor control [88](#)

EF Core class library

referencing [169](#), [170](#)

EF Core CLI tool

setting up [50](#), [51](#)

EF Core Fluent API

using, to define model [345](#)

end-of-life (EOL) [85](#)

entities

same scenario, using no tracking [359](#)

same scenario, using no tracking with identity resolution [360](#)

scenario, using default tracking [359](#)

tracking, controlling [357-359](#)

tracking, summary [360](#)

entity class [343](#)

entity creation

calculated properties on [354-356](#)

Entity Framework Core (EF Core) [341](#)

annotation attributes, using to define model [344](#), [345](#)

calculated properties, on entity creation [354-357](#)

conventions, using to define model [343](#)

data seeding [345](#)

models, scaffolding with existing database [342](#)

Northwind database model, querying [346-354](#)

used, for managing data [341](#), [342](#)

working with [342](#)

entity model [213](#), [343](#)

class library, creating for [51-53](#)

Entry control [88](#)

Express.js [429](#)

eXtensible Application Markup Language (XAML) [9](#), [79](#), [80](#)

converters type [82](#)

markup extensions [82](#), [83](#)

used, for simplifying code [80](#), [81](#)

F

FancyZones

reference link [576](#)

fanout exchange [526](#)

fault tolerance, with Polly

Circuit Breaker pattern [517](#)

MVC project, building to call faulty web service [519-524](#)

policies, applying to HTTP clients [518](#), [519](#)

policies, defining [517](#)

policies, executing [517](#)

random faults, adding to web service [519](#)

Retry pattern [516](#)

Retry pattern, implementing [524-526](#)

wait intervals, defining between retries [518](#)

Fluent API statements [343](#)

FluentAssertions

- making, about collections and arrays [224](#)
- making, about dates and times [224](#), [225](#)
- making, about equivalent objects [225](#), [226](#)
- making, about strings [222](#), [224](#)
- making, in unit testing [222](#)
- URL [226](#)

FluentValidation [227](#)

functions [390](#)

- adding [391-395](#)
- benefits [391](#)
- structured process [391](#)

G

Gemma3 models

- reference link [410](#)

General Availability [244](#)

General Data Protection Regulation (GDPR) [426](#)

Generative Pre-trained Transformer (GPT) [378](#)

Git [63](#)

GitHub [63](#)

GitHub Copilot

- for programmers [70](#), [71](#)
- responses, customizing [72](#), [73](#)

GitHub Copilot Chat [19](#)

GitHub repository [63](#)

- solution code, downloading from [64](#)

graphical user interface (GUI) [79](#)

graphics processing units (GPUs) [407](#)

GraphQL [6](#), [8](#), [591](#)

benefits, over OData [592](#)

capabilities [595](#)

field conventions [601](#), [602](#)

GraphQL client

ASP.NET Core MVC project, using as [623-628](#)

REST Client, using as [620-623](#)

GraphQL mutations [595](#)

implementing [638](#)

input [638](#)

payload [638](#)

GraphQL.NET [595](#)

URL [595](#)

GraphQL queries, EF Core models

support, adding [602](#), [603](#)

GraphQL query

arguments, specifying [594](#)

document format [592](#), [593](#)

executing [599](#), [600](#)

exploring, with Northwind [604-612](#)

fields requesting [593](#), [594](#)

filtering support, implementing [615-619](#)

filters, specifying [594](#)

naming [600](#)

paging support, implementing [612-614](#)

reference link [595](#)

selection set [592](#)

sorting support, implementing [619](#)

writing [599](#), [600](#)

GraphQL request formats

selecting [620](#)

GraphQL response format [620](#)

GraphQL schema

defining, for Hello World [597-599](#)

GraphQL service

mutations, adding [638](#), [640](#)

subscription, adding [649-651](#)

topic, adding [649-651](#)

GraphQL subscriptions [595](#), [649](#), [654](#)

benefits [654](#)

limitations [654](#)

grayscale thumbnails

generating [192-196](#)

Green Donut [596](#)

gRPC [6](#), [8](#), [659](#)

benefits [659](#)

contract, defining with .proto files [660-662](#)

limitations [659](#)

packages [663](#)

working [660](#)

gRPC faults

implementing [702](#)

gRPC, for EF Core model

gRPC client, implementing [678-681](#)

gRPC service, implementing [675-678](#)

implementing [675](#)

gRPC interceptors

client-side interceptor, adding [702-704](#)

exception handling [704](#)

implementing [702](#)

transient fault handling [704](#)

transient fault handling, adding [705-707](#)

gRPC JSON transcoding

enabling [707](#), [708](#)

gRPC-Web, comparing with [710](#)

implementing [707](#)

testing [709](#), [710](#)

gRPC method types [662](#)

bi-directional streaming method [663](#)

client streaming method [662](#)

server streaming method [662](#)

unary method [662](#)

gRPC service

improving, with native AOT publish [682-686](#)

gRPC service and client

building [663-673](#)

deadline for higher reliability, adding [688-691](#)

project file item configuration syntax [668](#), [669](#)

request and response metadata [686](#), [687](#)

testing [674](#), [675](#)

Grpc.Tools package [667](#)

gRPC-Web [660](#), [710](#)

GUIDs [304](#)

H

handlers [89](#)

Hangfire [191](#)

hosting, with website [548-553](#)

reference link [556](#)

replacing, with Quartz.NET [558](#)

used, for scheduling jobs [553-556](#)

headers exchange [527](#)

Hello World

GraphQL schema, defining for [597-599](#)

HorizontalAlignment property [126](#)

Hot Chocolate [595](#)

HTTP/1.1-based services [8](#)

HTTP/2

reference link [659](#)

HTTP caching

used, for caching web responses [512-516](#)

HttpRule class

reference link [707](#)

Hugging Face [406](#)

reference link [407](#)

Humanizer

case transformations [197](#)

dates and times, working with [205-207](#)

Singularize and Pluralize methods [198](#)

spacing conversions [198](#)

Humanizer third-party library [348](#)

humao.rest-client [19](#)

hybrid object caching [510](#)

enabling, in ASP.NET Core project [511](#), [512](#)

I

icsharpcode.ilspy-vscode [19](#)

identity resolution [357](#)

IdentityServer4 [486](#)

identity services [486](#)

JWT bearer authorization [487](#)

service clients, authenticating with JWT [488-490](#)

service clients, authenticating with JWT bearer authentication
[487](#)

ilspy-vscode [19](#)

images

drawing, with ImageSharp packages [196](#)

grayscale thumbnails, generating [192-196](#)

working with [191](#)

ImageSharp [189](#), [191](#)

used, for drawing images [196](#)

used, for working with web images [196](#)

inheritance hierarchies, with EF Core

mapping [360](#), [361](#)

mapping strategies, configuring [364](#), [365](#)

mapping strategies, example [365-374](#)

table-per-concrete-type (TPC) mapping strategy [363](#), [364](#)

table-per-hierarchy (TPH) mapping strategy [361](#), [362](#)

table-per-type (TPT) mapping strategy [362](#), [363](#)

in-memory caching

used, for caching objects [503-506](#)

Insiders edition [17](#)

integers [303](#)

Integrated Development Environments (IDEs) [14](#)

intermediate code (IL) [474](#)

International Organization for Standardization (ISO) [270](#)

invariant culture

outputting statistics [321](#), [322](#)

using temporarily [277](#)

iOS [10](#)

J

JSON Web Token (JWT) [487](#)

Just-In-Time (JIT) [474](#)

L

Language Service Protocol (LSP) host [18](#)

Large Language Models (LLMs) [4](#), [378](#)

access, obtaining to [380-384](#)

applications [378](#)

limitations [380](#)

working [378-380](#)

Large Language Model Meta AI (LLaMA) [407](#)

legacy .NET [1](#)

legacy Windows application platforms [8](#), [9](#)

legacy Windows platforms

modern .NET support [9](#)

Library Manager CLI [564](#)

ListView control [88](#)

live communication service, with SignalR

building [564](#)

server-side SignalR hub, enabling [565-570](#)

shared models, defining [564](#), [565](#)

Llama3 models

reference link [410](#)

LM Studio [415-418](#)

reference link [415](#)

local LLMs

Hugging Face [406](#)

LM Studio [415-418](#)

Ollama [407](#)

OllamaSharp with Microsoft.Extensions.AI [414](#)

running [406](#)

long-running background services

code execution, on timed schedule [546](#), [547](#)

implementing [539](#), [540](#)

job, scheduling with Hangfire [553-556](#)

queued messages, processing with worker service [542](#), [545](#)

website, building to host Hangfire [548-553](#)

worker service, building [540-542](#)

Long Term Support (LTS) [85](#), [342](#)

low-level APIs

data, managing with [310](#)

types, in ADO.NET [311](#), [312](#)

M

Mac build host

reference link [87](#)

macOS [10](#)

MapGroup [432](#)

mapping, between objects [213](#)

AutoMapper [213](#)

mappers, defining for AutoMapper configuration [215](#)

model, defining for AutoMapper configuration [213-215](#)

tests, performing for AutoMapper configuration [216-220](#)

mapping routes

reference link [431](#)

MapWhen [431](#)

Margin property [127](#), [128](#)

MCP server

building [398](#)

building, .NET used [400-403](#)

using, in code editor [403-405](#)

microservices [6](#)

Microsoft Learn

reference link [65](#)

Microsoft SQL Server [11](#), [12](#)

cons [13](#)

cost considerations [12](#)

performance and scalability considerations [12](#)

pros [13](#)

use case [12](#)

users [12](#)

mobile and desktop apps

.NET MAUI project, creating [92-95](#)

building, with .NET MAUI [90](#)

calculator page, using [104](#)

virtual Android device, creating for local app testing [90](#), [91](#)

Windows developer mode, enabling [92](#)

mobile development tools [85](#)

Model Context Protocol (MCP) [4](#), [398](#)

benefits [399](#)

client [399](#)

host [399](#)

server [399](#)

Model-View-Controller (MVC) [5](#)

Model-View-Update (MVU) [10](#)

modern databases [299](#)

connecting, to SQL Server database [301](#)

sample relational database [299](#), [300](#)

modern .NET [1](#)

support, for legacy Windows platforms [9](#)

MSBuild project tools [19](#)

ms-dotnettools.csdevkit [18](#)

ms-dotnettools.csharp [18](#)

ms-mssql.mssql [19](#)

MS SQL extension [45](#)

mutations [595](#), [638](#)

adding, to GraphQL service [638](#), [640](#)

exploring, with Nitro [640-642](#)

updates and deletes, implementing as [643-648](#)

MVC routing

versus Blazor routing [156](#)

MVC website

used, for sending messages to queue [530-534](#)

MySQL [11](#), [12](#)

cons [13](#)

cost considerations [12](#)

performance and scalability considerations [12](#)

pros [13](#)

use case [12](#)

users [12](#)

N

nanoservices [6](#)

native AOT

enabling, for project [476](#)

for ASP.NET Core [475](#), [476](#)

JSON serialization, enabling with [476](#)

limitations [474](#), [475](#)

project, building [477-482](#)

project, publishing [483-486](#)

reflection [475](#)

requirements [476](#)

startup time and resources, improving with [474](#)

network-bandwidth-constrained environment [8](#)

Neural Processing Unit (NPU) [409](#)

Nitro [595](#)

mutations, exploring [640-642](#)

Noda Time [191](#)

- concepts and defaults [289](#), [290](#)
- date/time types, converting between [291](#)
- exploring, in console app [291-294](#)
- reference link [296](#)
- unit testing and JSON serialization [296](#)
- using, with nanoseconds [294-296](#)
- working with [289](#)

Normalize method [294](#)

Northwind database [21](#), [300](#)

- creating, with SQL script [49](#), [50](#)
- GraphQL queries, exploring [604-612](#)
- SQL scripts [40](#)

Northwind database model

- querying [346-354](#)

no tracking [359](#)

NU1507 warning reference page

- reference link [35](#)

nuget.config

- reference link [37](#)

numbers

- working with [203](#), [204](#)

O

object-relational mapper (ORM) [334](#), [341](#)

OData [6](#), [8](#)

official .NET announcements

- reference link [65](#)

official .NET blog

reference link [65](#)

Ollama [407](#)

Ollama CLI [409](#), [410](#)

Ollama models [408](#), [409](#)

OllamaSharp

.NET package [411](#), [414](#)

with Microsoft.Extensions.AI [414](#), [415](#)

OpenAI

reference link [378](#)

OpenAI model

Agent Framework, using with [384](#)

OpenAPI

documentation link [429](#)

used, for testing web services [443-445](#)

P

Package Source Mapping (PSM) [35](#), [36](#)

enabling [36](#), [37](#)

reference link [37](#)

Padding property [128](#)

parameter mapping [433](#), [434](#)

PDF documents

generating, with class libraries [234](#), [235](#)

generating, with console apps [237-239](#)

platforms for deployment, .NET 10

Android [16](#)

iOS [16](#)

iPadOS [16](#)

Linux [16](#)

Mac [16](#)

Windows [16](#)

point-to-point real-time communication [8](#)

Polly

used, for fault tolerance [516](#)

Polyglot environment [8](#)

port

numbering conventions [21](#), [22](#)

POST [426-428](#)

PostgreSQL [11](#), [12](#)

cons [13](#)

cost considerations [12](#)

performance and scalability considerations [12](#)

pros [13](#)

use cases [12](#)

users [12](#)

PowerToys

reference link [576](#)

product and employee gRPC clients

adding [697-700](#)

product and employee gRPC services

implementing [696](#)

progress bar component

building [174](#), [175](#)

Project Reunion [9](#)

projects

managing [21](#)

naming conventions [21](#), [22](#)

properties, defining to reuse version numbers [24](#), [25](#)

structuring [21](#)

project templates [20](#), [21](#)

Protobuf [8](#), [659](#)

Protobuf style guide

reference link [661](#)

Protobuf VSC [19](#)

pseudo-classes

reference link [134](#)

public data service [8](#)

public service [8](#)

PUT [426-428](#)

Q

query string

parameters, setting [157](#)

QuestPDF

using, on Apple silicon Macs [234](#)

R

RabbitMQ

direct exchange [526](#)

fanout exchange [526](#)

headers exchange [527](#)

installation link [527](#)

message consumption, from queue with console app [535-539](#)

message, sending to queue with MVC website [530-534](#)

- queuing with [526](#), [527](#)
- reference link [527](#)
- setting up, with Docker [527-530](#)
- topic exchange [527](#)

Razor class libraries [5](#)

real-time communication

- history, on web [561](#)

Red Hat Enterprise Linux (RHEL) [14](#)

relational database

- using [299](#), [300](#)

Relational Database Management System (RDBMS) [299](#)

Reliable Web App (RWA) pattern

- reviewing [557](#)

Remote Procedure Call (RPC) [7](#), [562](#), [659](#),

Request Delegate Generator (RDG) [476](#)

REST Client [19](#), [446](#)

- using, as GraphQL client [620-623](#)

ResX Resource Manager [288](#)

Retry pattern [516](#)

- implementing, for transient fault handling [524-526](#)

return values [434](#)

- multiple typed outcomes, returning [435](#)

Rider [15](#)

S

same origin policy [451](#)

satellite assemblies [278](#)

Scalable Vector Graphics (SVG) [168](#)

secrets

storing, alternatives [337](#)

SELECT

documentation link [307](#)

Serilog [189](#)

used, for rolling file [209-211](#)

server interactivity

used, for building component [172-174](#)

serverless nanoservice [8](#)

Server-Sent Events (SSE) [649](#)

server streaming method [662](#)

service [6](#)

building, that supports GraphQL [596](#), [597](#)

service architecture [495](#)

numbers, scaling [496](#)

system, slowest parts [495](#), [496](#)

service technologies [4](#), [5](#)

cons [7](#)

pros [7](#)

service-to-service [8](#)

shared resources

changing, dynamically [106-114](#)

defining, at app level [104](#)

referencing [106](#)

using [104](#)

Shell control [87](#)

shell navigation

adding [96-101](#)

Short Term Support (STS) [85](#)

SignalR [6](#), [8](#), [561](#), [562](#)

client platforms [563](#)

method signatures, designing [563](#)

single page application (SPA) frameworks [152](#)

Skeet, John [297](#)

Skia [121](#), [122](#)

specialized libraries [3](#)

specialized type handling

testing [701](#), [702](#)

SQL script

Northwind database, creating [49](#), [50](#)

SQL Server

connecting, from Visual Studio [47](#)

connecting, from VS Code [48](#), [49](#)

SQL Server authentication

user and password, setting [58](#)

SQL Server container

running, with user interface [44](#)

SQL Server container image

running [42-44](#)

setting up [41](#)

SQL Server database

connecting to [301](#), [302](#)

SQL Server Data Tools (SSDT) [45](#)

SQL Server Management Studio (SSMS) [45](#)

SQL Server (mssql) for VS Code [19](#)

Stack Overflow

reference link [378](#)

Standard Term Support (STS) [342](#)

static server rendered component

building, for data [170](#), [172](#)

static server rendering (SSR) [152](#)

static site generator (SSG) [152](#)

stored procedures

executing, with ADO.NET [323-326](#)

Strawberry Shake [596](#)

console app client, creating [633-637](#)

URL [633](#)

streams [582](#)

structured event data logging [207](#)

console, logging to [209](#), [211](#)

file, rolling with Serilog [209-211](#)

sensitive data logging, avoiding [211](#), [212](#)

Serilog sinks [208](#)

structured event data [207](#)

Structured Query Language (SQL) [302](#)

styling controls, Avalonia [131](#)

defining [131](#)

global styles [134](#)

style inheritance [132](#)

style selectors [132](#)

style triggers [133](#), [134](#)

subscriptions [595](#)

adding, to GraphQL service [649-651](#)

Swashbuckle [190](#)

SwiftUI [10](#)

T

table-per-concrete-type (TPC) mapping strategy [363](#), [364](#)

table-per-hierarchy (TPH) mapping strategy [361](#), [362](#)

table-per-type (TPT) mapping strategy [362](#), [363](#)

text

working with [196](#), [197](#)

third-party libraries [189](#), [190](#)

packages [190](#), [191](#)

TimeOnly

working with [256](#)

time provider

implementing [260](#), [262](#)

used, for unit testing [258-260](#)

TimeZoneInfo class [264-269](#)

time zones

DateTime class [263](#)

TimeZoneInfo class [263](#), [264](#)

working with [263](#)

tintoy.msbuild-project-tools [19](#)

tokens [379](#)

tools

disabling [73](#)

topic

adding, to GraphQL service [649-651](#)

subscribing to [651-653](#)

topic exchange [527](#)

training process

backward pass and optimization [379](#)

forward pass [379](#)

loss calculation [379](#)

Transact-SQL (T-SQL)

data, managing with [302](#)

data types [303](#)

data types, specifying [305](#)

documenting, with comments [304](#)

flow, controlling [305](#)

operators [305](#)

reference link [303](#)

variables, declaring [305](#)

Transformer [379](#)

attention mechanisms [379](#)

feed-forward neural networks [379](#)

T-SQL data types [303](#)

for primary key [303](#), [304](#)

primary key options [304](#)

U

unary method [662](#)

Uniform Resource Locators (URLs) [5](#)

unit testing

with time provider [258-260](#)

Universal Windows Platform (UWP) [8](#), [9](#)

Uno

URL [10](#)

unpkg [564](#)

upsert operation [426](#)

user interface

localizing [278](#)

V

VerticalAlignment property [127](#)

view models [213](#)

virtual Android device

creating, for local app testing [90](#), [91](#)

Visual Basic [9](#)

Visual Studio [14](#)

downloading [16](#), [17](#)

installing [16](#), [17](#)

keyboard shortcuts [17](#)

VPS (Virtual Private Server) [60](#)

VS Code

downloading [17](#), [18](#)

for cross-platform development [14](#)

installing [17](#), [18](#)

keyboard shortcuts [20](#)

versions [19](#)

VS Code extensions

installing [18](#), [19](#)

managing, at command prompt [19](#)

W

warnings

treating, as errors [38](#), [39](#)

web client, with SignalR JavaScript library

building [570](#)

chat feature, testing [575-578](#)

chat page, adding to MVC website [571-574](#)

Web Forms [9](#)

web services [423](#)

acronyms [423](#), [424](#)

documenting, with OpenAPI [428](#)

documenting, with Swagger [428](#)

documenting, with Swashbuck [428](#)

GET request and common responses [424](#), [426](#)

HTTP requests and responses [424](#)

testing, with OpenAPI [443-445](#)

websites [5](#)

WebSockets (WS) [424](#), [562](#)

reference link [562](#)

Win32 API [8](#)

Windows [10](#)

Windows App SDK [8](#), [9](#)

reference link [9](#)

Windows Communication Foundation (WCF) [7](#), [9](#)

Windows Compatibility Pack

reference link [10](#)

Windows developer mode

enabling [92](#)

Windows Forms [8](#), [9](#)

Windows-only apps [8](#)

Windows Presentation Foundation (WPF) [8](#), [9](#), [80](#), [120](#)

Windows Service [540](#)

Windows Store [8](#)

Windows Workflow (WF) [9](#)

WinUI 3 [9](#)

workloads [5](#)

X

Xamarin

version support [85](#)

Xamarin.Forms [80](#)

XMLHttpRequest [561](#)

xUnit [60](#)

benefits [60](#)

class libraries, testing [60-63](#)